MOST ToGo Architecture and Implementation

User's Guide

Supporting **MOST**® Networks

Media Oriented Systems Transport

© 2014 Microchip Technology Inc.

MAY 2014

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE**.** Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

Trademarks

The Microchip name and logo, the Microchip logo, dsPIC, FlashFlex, KEELOQ, KEELOQ logo, MPLAB, PIC, PICmicro, PICSTART, PIC^{32} logo, rfPIC, SST, SST Logo, SuperFlash and UNI/O are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

FilterLab, Hampshire, HI-TECH C, Linear Active Thermistor, MTP, SEEVAL and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Silicon Storage Technology is a registered trademark of Microchip Technology Inc. in other countries.

Analog-for-the-Digital Age, Application Maestro, BodyCom, chipKIT, chipKIT logo, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, dsSPEAK, ECAN, ECONOMONITOR, FanSense, HI-TIDE, In-Circuit Serial Programming, ICSP, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, mTouch, Omniscient Code Generation, PICC, PICC-18, PICDEM, PICDEM.net, PICkit, PICtail, REAL ICE, rfLAB, Select Mode, SQI, Serial Quad I/O, Total Endurance, TSHARC, UniWinDriver, WiperLock, ZENA and Z-Scale are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

GestIC and ULPP are registered trademarks of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

flexPWR, JukeBlox, Kleer, KleerNet, MediaLB, and MOST

The preceding is a non-exhaustive list of trademarks in use in the US and other countries. For a complete list of trademarks, email a request to legal.department@microchip.com. The absence of a trademark (name, logo, etc.) from the list does not constitute a waiver of any intellectual property rights that SMSC has established in any of its trademarks.

All other trademarks mentioned herein are property of their respective companies.

© 2014, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

ISBN: 978-1-62077-895-1

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

© 2014 Microchip Technology Inc.

Preface

NOTICE TO CUSTOMERS

All documentation becomes dated, and this manual is no exception. Microchip tools and documentation are constantly evolving to meet customer needs, so some actual dialogs and/or tool descriptions may differ from those in this document. Please refer to our web site (www.microchip.com) to obtain the latest documentation available.

Documents are identified with a "DS" number. This number is located on the bottom of each page, in front of the page number. The numbering convention for the DS number is "DSXXXXXA", where "XXXXX" is the document number and "A" is the revision level of the document.

For the most up-to-date information on Microchip development tools, please visit www.microchip.com.

INTRODUCTION

This chapter contains general information that will be useful to know before using the *MOST ToGo Architecture and Implementation*. Items discussed in this chapter include:

- Notice to Customers
- Introduction
- Document Layout
- Conventions Used in this Guide
- Warranty Registration
- The Microchip Website
- Customer Change Notification Service
- Customer Support
- Recommended Reading
- Document Revision History

© 2014 Microchip Technology Inc.

DOCUMENT LAYOUT

This Specification describes how to use the *MOST ToGo Architecture and Implementation*. The document is organized as follows:

- **Chapter 1. “Overview”**
- **Chapter 2. “MOST ToGo Architecture”**
- **Chapter 3. “Device Implementation”**
- **Appendix A. “Glossary and General Terms”**

CONVENTIONS USED IN THIS GUIDE

Within this manual, the following abbreviations and symbols are used to improve readability.

Example	Description
BIT	Name of a single bit within a field
FIELD.BIT	Name of a single bit (BIT) in FIELD
x…y	Range from x to y, inclusive
BITS[m:n]	Groups of bits from m to n, inclusive
PIN	Pin Name
SIGNAL	Signal Name
msb, lsb	Most significant bit, least significant bit
MSB, LSB	Most significant byte, least significant byte
zzzzb	Binary number (value zzzz)
0xzzz	Hexadecimal number (value zzz)
zzh	Hexadecimal number (value zz)
rsvd	Reserved memory location. Must write 0, read value indeterminate
`code`	Instruction code, or API function or parameter
Multi Word Name	Used for multiple words that are considered a single unit, such as: *Resource Allocate* message, or *Connection Label*, or *Decrement Stack Pointer* instruction.
Section Name	Emphasis, Reference, Section or Document name.
$\overline{\text{VAL}}$	Over-bar indicates active low pin or register bit
x	Don’t care
<Parameter>	<> indicate a Parameter is optional or is only used under some conditions
{,Parameter}	Braces indicate Parameter(s) that repeat one or more times.
[Parameter]	Brackets indicate a nested Parameter. This Parameter is not real and actually decodes into one or more real parameters.

© 2014 Microchip Technology Inc.

WARRANTY REGISTRATION

Please complete and mail the Warranty Registration Card that was enclosed with the development board. Sending in the registration card entitles you to receive new product updates. Interim software releases are available at the Microchip web site.

THE MICROCHIP WEBSITE

Microchip provides online support via our WWW site at www.microchip.com. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQ), technical support requests, online discussion groups, Microchip consultant program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

CUSTOMER CHANGE NOTIFICATION SERVICE

Microchip's customer notification service helps keep customers current on Microchip products. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, access the Microchip web site at www.microchip.com. Under "Support", click on "Customer Change Notification" and follow the registration instructions.

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or Field Application Engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document.

Technical support is available through the web site at: http://microchip.com/support

RECOMMENDED READING

This user's guide describes how to use *MOST ToGo Architecture and Implementation* and references the following documents as recommended and supplemental resources.

Documents listed below and referenced within this publication are current as of the release of this publication and may have been reissued with more current information. To obtain the latest releases of Microchip documentation please visit the Microchip website. Please note, some Microchip documentation may require approval. Contact information can be found at www.microchip.com.

All non-Microchip documentation should be retrieved from the applicable website locations listed below. Microchip is not responsible for the update, maintenance or distribution of non-Microchip documentation.

Because the Internet is a constantly changing environment, all Internet links mentioned below and throughout this document are subject to change without notice.

[1] OS81092 Evaluation Board User's Guide
DS60001239. Microchip. www.microchip.com.

[2] MOST Specification 3.0
MOST Cooperation. www.mostcooperation.com.

[3] MOST System Management Module (MSMM)
MOST Cooperation. www.mostcooperation.com.

[4] MOST Editor User Manual 3V2
How to use the MOST Editor to create and edit function blocks.
MOST Cooperation. www.mostcooperation.com.

[5] K2L Automotive Test System Tool (ATS)

[6] Port Message Viewer v6+
DS60001219. Microchip. www.microchip.com.

[7] MOST Dynamic Specification
Describing the dynamic behavior of MOST devices.
MOST Cooperation. www.mostcooperation.com.

[8] INIC Hardware Concepts
DS60001264. Microchip. www.microchip.com. General guide for interfacing INIC to the external host controller (EHC), to the Physical Layer, and to power management circuitry for MOST25, MOST50 and MOST150

[9] MOST ToGo Hardware Principles
DS20005241. Microchip. www.microchip.com. The INIC Hardware Concepts applied to the specific case of the *MOST ToGo* basic system.

[10] MPM85000 Automotive Power Management Device Data Sheet
DS85000AP4. Microchip. www.microchip.com.

[11] MOST Electrical Control Line Specification
MOST Cooperation. www.mostcooperation.com.

[12] MOST ToGo Code Documentation
Microchip. www.microchip.com.

[13] MOST NetServices Layer I User’s Guide
Microchip. www.microchip.com.

[14] MOST NetServices Layer II User’s Guide
Microchip. www.microchip.com.

© 2014 Microchip Technology Inc.

DOCUMENT REVISION HISTORY

MOST ToGo Architecture and Implementation.The most extensive and pertinent application changes are listed in Table 1, although various other differences may be observed between document revisions.

TABLE 1: REVISION SUMMARY

	Description of Changes
DS60001272A:	5/7/14 -- Initial Release for MOST ToGo Evaluation Kit Launch

Chapter 1. Overview

The *MOST ToGo* system is meant to be an exemplary project with a focus on teaching how to create a networked infotainment system based on MOST technology. As such, the system architecture is kept simple, and the function of each device is kept as simple as possible to aid in learning. The basic *MOST ToGo* system contains 3 nodes: A simple audio source device (`AuxIn`), a simple audio sink device (`Amp`), and a controller device that has a simple user interface (`HMI`). The HMI is the network master, while the source and sink devices are network slave devices.

The basic system described here is meant to be a base system that will be expanded through future projects to include more examples with which to create more sophisticated networks. These projects will add video sources and displays, DTCP encryption/decryption, TCP/IP over a MOST network, module update by flashing over a MOST network, remote control applications, CAN gateway, and wireless audio. But first, understanding this basic system is in order before increasing the complexity.

The devices are all based on the same OS81092 ePhy Evaluation Board, (OS81092 Evaluation Board User's Guide [1]), only the firmware between the devices is different in order to provide the different functionality. The evaluation board has more features than needed for this first basic example in order to facilitate these future projects. A MOST150 version of these boards will also be available in the future.

This document describes the system architecture, and the firmware architecture used to implement the system. While it is a simple system, there is plenty to learn from it including: functional partitioning, creating function catalogs, implementing MOST NetServices™, creating function blocks (FBlocks), implementing a network master and connection master, implementing full MOST network compliant power management, and implementing diagnostics.

1.1 AUDIENCE

It is assumed that the reader is familiar with MOST networks and the MOST Specification 3.0 [2], and has ideally had MOST Foundation Training. The reader should understand the three major data channels on the MOST network: Control Channel for messaging between nodes (similar to CAN messages), Asynchronous Channel for packet type data (similar to Ethernet messages), and Synchronous Channels for zero overhead 100% Quality of Service (QOS) transmission of audio and video data (unique to MOST networks) The reader should also have some familiarity with the Microchip INIC (Intelligent Network Interface Controller) devices that are the interface between an application and the physical network. The first section is intended for system architects while the later sections will be of more interest to system implementers.

© 2014 Microchip Technology Inc.

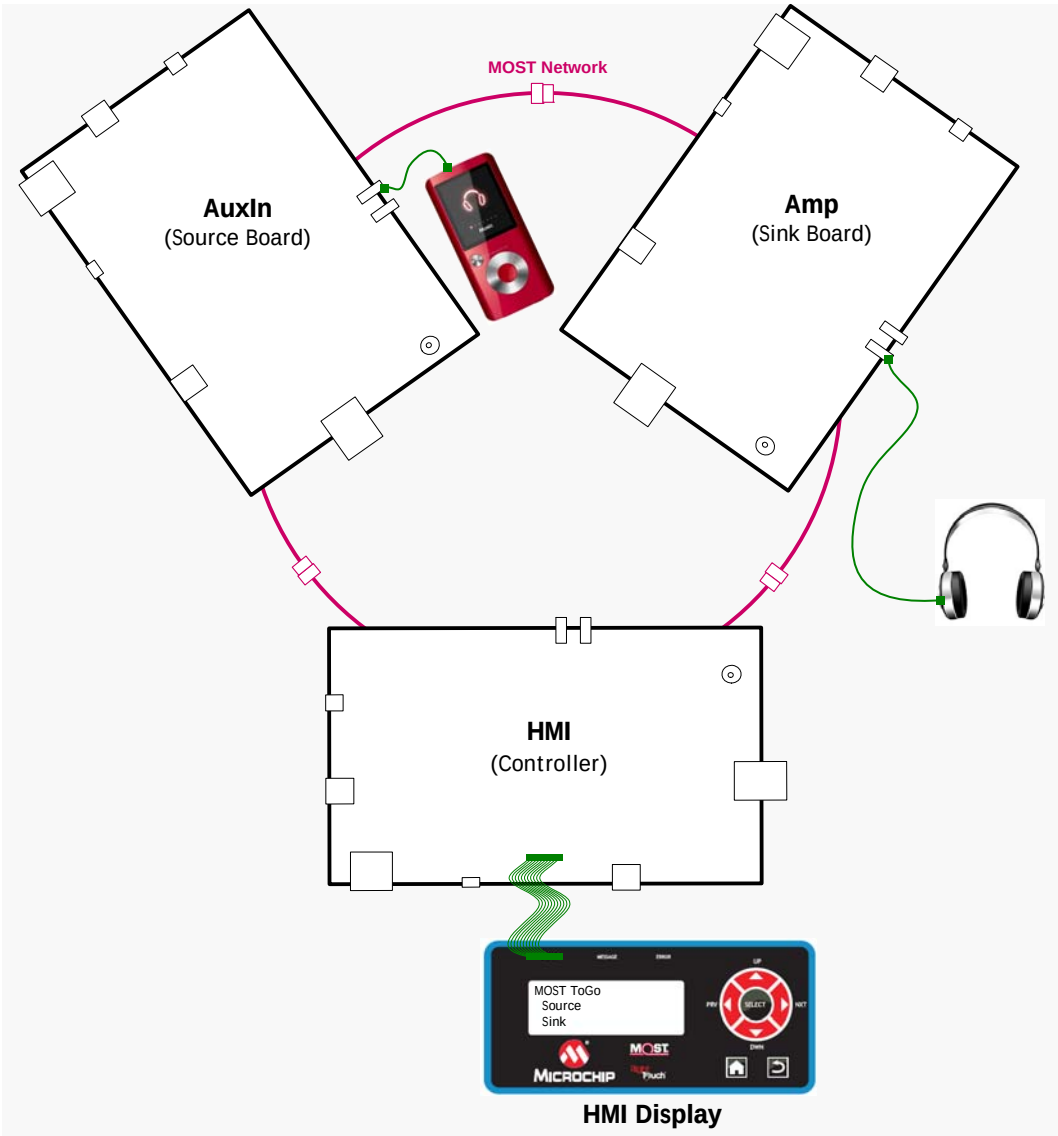
Chapter 2. *MOST ToGo* Architecture

With any infotainment system and hence any MOST system, the first step is to decide what the functionality of the whole system should be. Usually, when an OEM starts a system design, it is not just a single system, but a range of systems to apply to luxury, mid-level, and base platforms. This leads to a range of devices that are put together in different configurations for the different platforms. The temptation is to always create the most fully featured (expensive) system first as those platforms can better absorb the cost, and modules created can 'trickle down' to the more basic systems. We would caution against this approach as it similar to trying to build a space shuttle before successfully getting a single engine prop plane off the ground. Consider this basic *MOST ToGo* design the very base system, and the appropriate starting place for a first MOST system.

For the basic *MOST ToGo* system we want to show three major aspects of a MOST network, a source device (audio source), a sink device (audio sink), and a controller device to setup and manage the network. If each function is in a separate device, or node, then this will lead to a 3 node network with 2 slave devices and a master node. This system should also conform to the latest MOST Specification 3.0 [2] for MOST50 and MOST150. The system should also support full MOST network power management including wakeup and shutdown and diagnostics in the event of an unlocked ring. Once these basics are understood and implemented, adding additional functionality such as video and packet data becomes a much easier task as will be shown in future add on projects for the *MOST ToGo* system. Each node will use the MOST NetServices which is provided by Microchip. NetServices is a middleware software stack for MOST systems that provides similar functionality of a TCP/IP stack for an Ethernet controller, i.e. it abstracts the functionality of the INIC and provides an API to the application to operate the network. The basic system is shown in Figure 2-1 below.

© 2014 Microchip Technology Inc.

FIGURE 2-1: *MOST ToGo* BASIC SYSTEM BLOCK DIAGRAM



2.1 PARTITIONING THE FUNCTIONALITY

To create a system as outlined above, the functionality must be partitioned into manageable pieces, the engineering approach of divide and conquer. The MOST technology paradigm for this division is to use *Function Blocks*, referred to as *FBlocks* in a MOST system, and represented by an *FBlockID* in MOST network messages. FBlocks are like classes in an object oriented programming language. The MOST Specification 3.0 [2] defines more than 40 FBlocks to use as a starting point. If functionality is required that is not met by these FBlocks, then the user is free to create new ones. Note that it is possible (indeed likely) that there may be 2 instances of the same function block in a system (i.e. 2 amplifiers or 2 displays). To prevent confusion over which FBlock should be addressed, an instance ID (InstID) is assigned to each FBlock. The combination of FBlockID and InstID is called the *functional address* and must be unique in the system. In an object oriented language, when an object of a class is instantiated, that object

© 2014 Microchip Technology Inc.

is given a name that is unique in the program. Consider the functional address (the combination of FBlockID and InstID) to be the unique name of a particular instance of a FBlock.

Each FBlock is composed of *Functions* which can represent *Methods* or *Properties*, just like methods and properties of a class, and are represented by *Function IDs* (*FktID* – German abbreviation) in MOST network messages. Each function supports various operation types (*OpType*) such as *Set* or *Get* for a property or *Start* for a method.

The four elements described above, FBlockID, InstId, FktId, and OpType form the backbone of how MOST network functions are defined and how MOST network messages are constructed. In fact, the form of a MOST network message is:

`FBlockID.InstId.FktId.OpType.Data….`

Where the Data field contains any parameters the function may need.

The MOST Specification 3.0 [2] defines several categories of FBlocks and assigns them to a range of FBlockIDs. Some examples of the common types are:

- Administration
  - `NetBlock`
  - `NetworkMaster`
  - `EnhancedTestability`
- Audio
  - `AudioAmp`
  - `HeadphoneAmp`
  - `AuxiliaryInput`
  - `MicrophoneInput`
- Drives
  - `AudioDiskPlayer`
  - `DVDPlayer`
- Receiver
  - `AM/FMTuner`
  - `TVTuner`
  - `DABTuner`
- Communications
  - `Telephone`
  - `PhoneBook`
  - `NavigationSystem`
- Video
  - `VideoDisplay`
  - `VIdeoCamera`

This list is just a few of the FBlocks defined by the MOST Cooperation, which maintains the specification for each. Again, if a function is needed that is not listed, a new FBlock can be created, and there are ranges of FBlockIDs reserved for this purpose. Similarly, if a particular FBlock does not contain a function that is needed, new functions can be defined and added to an existing FBlock.

© 2014 Microchip Technology Inc.

For the *MOST ToGo* basic system, the source will use FBlock `AuxiliaryInput` (FBlock `AuxIn`), the sink will use FBlock `AudioAmplifier` (FBlock `AudioAmp`), while the master device will need FBlock `NetworkMaster`, FBlock `ConnectionMaster` and FBlock `HMI`. All nodes are required to have two other administrative FBlocks, `NetBlock` and `EnhancedTestability`.

2.2 FBlock OVERVIEW

Below is a short description of each function block needed by the system. The firmware implementation sections contain a description of how the FBlocks are implemented using the NetServices command interpreter.

2.2.1 NetBlock

This is an administrative function block required in all nodes. It is used to manage network specific properties such as Node Address, FBlock IDs (a report of all FBlocks on the device), and functions for use in power management such as network shutdown. The Network Master relies on the `NetBlock` of each device to control the network, discover the functionality of each device, and for power management. Since this is a required FBlock for all nodes, the NetServices software stack implements it internally. There are a few functions that require input from the application, so NetServices does a callback to the application so it can supply the data.

2.2.2 Enhanced Testability (FBlock ET)

This is another administrative function block required in all nodes. This is used by an external test tool to verify compliance to the MOST Specification 3.0 [2]. As with FBlock `NetBlock`, NetServices implements this FBlock internally and uses callbacks to the application to get data or perform actions that NetServices cannot do on its own, for example to reset the processor, or shutdown the network.

2.2.3 Auxiliary Input (FBlock AuxIn)

This function block is the interface between the analog input jack on the board and its associated A/D converter (CODEC) and a streaming channel on the network. The major functions of FBlock `AuxIn` are *Allocate* (put the streaming data onto a network channel) and *DeAllocate* (stop the streaming data and release the network channel) as well as *Mute* the channel. This function block is implemented by the application and requires direct communication with the INIC to perform its functions.

2.2.4 Audio Amplifier (FBlock AudioAmp)

This function block is the complement to FBlock `AuxIn`. It is the interface between streaming data on the network and the on board D/A converter (CODEC) and its associated analog output jack (headphone or powered speakers). The major functions of FBlock `AudioAmp` are *Connect* (send streaming data from a network channel to the CODEC), and *Disconnect* (disconnect from the network channel, stopping the flow of data to the CODEC) Also FBlock `AudioAmp` has functions for *Volume*, *Bass*, *Treble*, and *Mute*. This function block is implemented by the application and requires direct communication with the INIC and to the CODEC to perform its functions.

© 2014 Microchip Technology Inc.

2.2.5 Network Master (FBlock NetworkMaster)

Every MOST system must have one node that is the network master. It must be both the timing master, which is a hardware function of INIC, and the Network Master which is a function of network administration. The timing master is responsible for sending out the network frames which are passed around the network, and ensuring the integrity of the frames as they come back in to the master node. This is all handled by INIC in hardware and is just a register setting; all INICs are capable of being the timing master.

FBlock `NetworkMaster` is software functionality on top of the timing master that checks for proper configuration of all devices in the network. As soon as the network starts up, the Network Master must perform a system scan in which the Network Master asks each node for its function blocks. When a node returns its implemented function blocks (FBlock IDs), it also returns its node address. The Network Master catalogs all function blocks and their associated node addresses.

This catalog of node addresses and function blocks is called the *Central Registry*. The Network Master checks the registry for consistency; no duplication of node addresses or functional addresses (the combination of the FBlockID and InstID) is allowed. If the network master detects a conflict, it will command the offending device to change its node address or InstID to resolve the conflict. Once the network master has determined there are no conflicts, it broadcasts a message to the network that the system is OK (`NetworkMaster.ConfigStatus.Status.OK`). At this time all nodes know that the network is ready for normal operation.

FBlock `NetworkMaster` is an administrative function block, but it is not implemented directly by NetServices since only one node in the system needs this function. Microchip does offer the MOST System Management Module (MSMM) [3] which is an additional middleware layer that sits on top of NetServices and does provide the FBlock `NetworkMaster`. It is highly configurable and provides all of the functionality the MOST Specification 3.0 [2] defines for a network master.

2.2.6 Connection Master (FBlock ConnectionMaster)

FBlock `ConnectionMaster` is another administrative function that is only required in one node. Its function is to control the synchronous (audio and video) connections on the network. The application itself must contain the logic of what should be connected at any given moment. For example, does the user want to listen to FM radio or MP3 Player? When a phone call comes in, should the audio be muted in the whole vehicle or just the front seat? Should the display be showing the rear view camera or the navigation display? These decisions are made by the application logic. In the *MOST ToGo* system, that logic is in a module called the AV_Manager (avm.c). All systems need such logic, although the functionality could be split among several modules.

The Connection Master takes requests to connect or disconnect a source and a sink of the same data type, (e.g. audio or video). For example, in the *MOST ToGo* system the AV_Manager might ask the Connection Master to connect the AuxIn channel of the source node to the AudioAmp channel of the sink node. The Connection Master will determine if the AuxIn data is currently being streamed on the network, and if not, ask the AuxIn device to *Allocate* its source to get the data streaming on the network. When the AuxIn device replies to the Connection Master with its channel number (when the allocation is complete), then the Connection Master will instruct the AudioAmp device to *Connect* to this

© 2014 Microchip Technology Inc.

channel. When the AV_Manager requests the Connection Master to remove the connection, it will instruct the sink (AudioAmp) to *DisConnect*, then it will check to see if any other sinks are using the AuxIn source, and if not, it will instruct the AuxIn source to remove its streaming data from the network (*DeAllocate*). The Connection Master maintains a *Connection Table* which keeps track of all sources and sinks in the network and their current connection status. Other devices can query this table if needed.

The FBlock `ConnectionMaster` is not provided by NetServices, but is included with the MSMM stack available from Microchip. Since the Connection Master makes heavy use of the Central Registry maintained by the Network Master, it is highly recommended that the FBlock `ConnectionMaster` reside on the same device as the Network Master.

2.2.7 Human Machine Interface (FBlock HMI)

FBlock `HMI` handles inputs from the user (switches, touch screen, voice) and communicates these inputs to the application logic. The application logic will determine what should be displayed based on the state of the system and use FBlock `HMI` to control the displays, lights etc. Since this function block is highly dependent on the hardware of the system, it must be implemented by the user. While all of the logic of the user interface could be in this module, it is recommended that most of the logic be incorporated into a user interface module (e.g. UI_Manager) and have FBlock `HMI` mostly concerned with controlling the hardware (switches and displays). In the *MOST ToGo* system that logic is in a module called the HMI_Manager (*hmi_manager.c*).

The *MOST ToGo* user interface consists of only some switches (RightTouch™ cap sense inputs) a 4 line LCD display, and a few LEDs. Therefore FBlock `HMI` on the *MOST ToGo* system has only a few functions such as ButtonStatus, ClearLCD, WriteLCD etc. ButtonStatus is used whenever any button of the user interface is pressed. The HMI then passes that information on to the HMI_Manager module and logic there determines what to do with the button event.

Note that the look and feel of the entire system is essentially controlled by FBlock `HMI`, the HMI_Manager, and the AV_Manager. These modules are all done by the user (OEM or Tier 1 supplier) and are the elements that make a system unique – easy and intuitive to use or (sometimes) confusing and over complicated. This is the domain of expertise provided by the OEM or their chosen Tier 1 supplier. The MOST Specification, NetServices, or MSMM do not specify or dictate how that part of the application is done. Part of the goal of a MOST system is to separate this look and feel logic from the nuts and bolts of network management, message distribution, and source and sink control.

2.3 FBlock SHADOWS

FBlocks and controllers (like the AV_Manager) work together like client – server pairs on a PC network. The FBlock is the server, providing some functionality and having properties the client (controller) can access. When a controller like the AV_Manager sends a message to an FBlock like `AudioAmp` to return its current volume for example, *Amp.InstID.Volume.Get*, FBlock `HMI` `AudioAmp` will send a message back to the controller (whoever asked) returning the current setting with a status message like *Amp.InstID.Volume.Status.12*, indicating the current volume is 12. The controller must ‘catch’ this message to get the volume setting, but the controller device may not have an FBlock `AudioAmp`, so it would have no entries in its command interpreter table to accept messages with an

© 2014 Microchip Technology Inc.

FBlock ID of FBlock AudioAmp. So, each controller implements what is called an FBlock shadow in MOST system terminology to enter an FBlock in its own command interpreter table to accept status and error messages from FBlocks that it controls across the network. As status messages arrive in response to commands, they get routed by the FBlock shadow to the controller where it can read and save the current value of the properties of the remote FBlock. Over time, the controller builds a complete copy or 'shadow' of the properties that exist on the remote FBlock.

In the basic *MOST ToGo* system, the master node has the AV_Manager and Connection Master which act together as the controller for FBlock AuxIn and FBlock AudioAmp. Therefore, the master device will need FBlock shadows for AuxIn and AudioAmp. Less obvious is that the Network Master function block may send messages to slave devices (e.g. NetworkMaster.ConfigStatus.OK) that they need to 'catch'. Therefore, all slave devices need an FBlock shadow for FBlock NetworkMaster. This shadow concept will come in to clearer focus in the implementation sections where it can be seen how they are implemented and how they route message responses to the controllers.

2.4 WORKING TOGETHER

Now that the basics of FBlocks have been introduced, a couple of examples of how they all work together to accomplish system functionality will help make the use of function blocks more clear.

Consider the user pressing the HOME key on the user interface (LCD Display w/ RightTouch buttons). The hardware handler detects that a button is pressed and sends a message to FBlock HMI which then qualifies the button, and calls the HMI_Manager to handle the key. The reason FBlock HMI is used instead of the hardware handler just calling UI_Manager directly is that the button press could have occurred on another module, such as a rear seat display. Having the hardware handler send messages to FBlock HMI means the user interface or FBlock HMI could exist anywhere on the network. The UI_Manager will receive which button was pressed as a parameter and recognize it as a HOME request. The UI_Manager can handle this request itself. It will update the state of its menu system, and refresh what should be displayed. Then the UI_Manager will send messages to FBlock HMI to display the new proper menu. Again FBlock HMI is used to update the display instead of calling the hardware functions directly because the HMI could be on a different node, or there could be multiple HMIs to update.

Now consider that the menu is setup to select a new source (AuxIn) to listen to (i.e. send to AudioAmp), and the user presses the SELECT button. Again the hardware handler detects the button press and sends a message to FBlock HMI. FBlock HMI then calls the UI_Manager with the button as a parameter. Based on the state of the menu system, the UI_Manager recognizes the user is requesting to listen to a particular source. The UI_Manager calls the AV_Manager with parameters indicating the user is requesting a connection to a particular source. The AV_Manager can consider the request, make sure it is valid, and allowed at this time (e.g. phone is not in use). If it is allowed, then the AV_Manager will build a message to FBlock ConnectionMaster containing FBlock and InstID of both the source and the sink. The AV_Manager looks up this information from a table it maintains based on inputs from the Connection Master. The Connection Master receives this message, and builds messages for the source, AuxIn in this case, and then for the sink (AudioAmp). As the AuxIn node replies that it has put its source on the network, and which channel it is on (response received via FBlock AuxIn shadow), the Connection Master updates its internal tables. Then it will

© 2014 Microchip Technology Inc.

instruct the AudioAmp to connect to the channel reported by AuxIn. When the AudioAmp responds (again, response received via FBlock AudioAmp shadow) that the connection is done, the Connection Master will inform the AV_Manager (who requested the connection) that the job is done. The AV_Manager will update its internal tables and then inform the UI_Manager that the job is done. The UI_Manager can then update the display image accordingly and send commands to FBlock HMI to update the display(s).

This may sound like a convoluted chain of events, but in reality in the *MOST ToGo* system, from the time the button is pressed until the audio is playing and the display is updated is between 50 and 100 ms, fast enough to sound and look real time to a user. The advantage of this approach is that all of the application logic is centralized in the UI_Manager and AV_Manager modules, while the resources can be distributed anywhere on the network. For example, the source does not have to be concerned about the state of the system, and whether it is valid or not for it to be playing – that is determined by the AV_Manager, the source simply does what it is told without any concern about the rest of the system.

The FBlocks provide excellent encapsulation of the functionality of the system with a clear API for each block that is activated by sending MOST network messages to the FBlock. The messages are defined by the structure of the FBlock itself when it is designed. The functions of an FBlock are testable while even being isolated from the rest of the system. Services within the NetServices middleware stack assure delivery of these messages no matter where the FBlock resides on the network. FBlocks located on the same device communicate with each other in exactly the same fashion as those located across the network. In fact an FBlock can be moved from one device to another with no change in the application logic.

2.5 FBlock SUMMARY

The basic *MOST ToGo* system that has been described can be implemented using just the FBlocks outlined above. Now these FBlocks need to be partitioned across the 3 devices. This is shown in the table below.

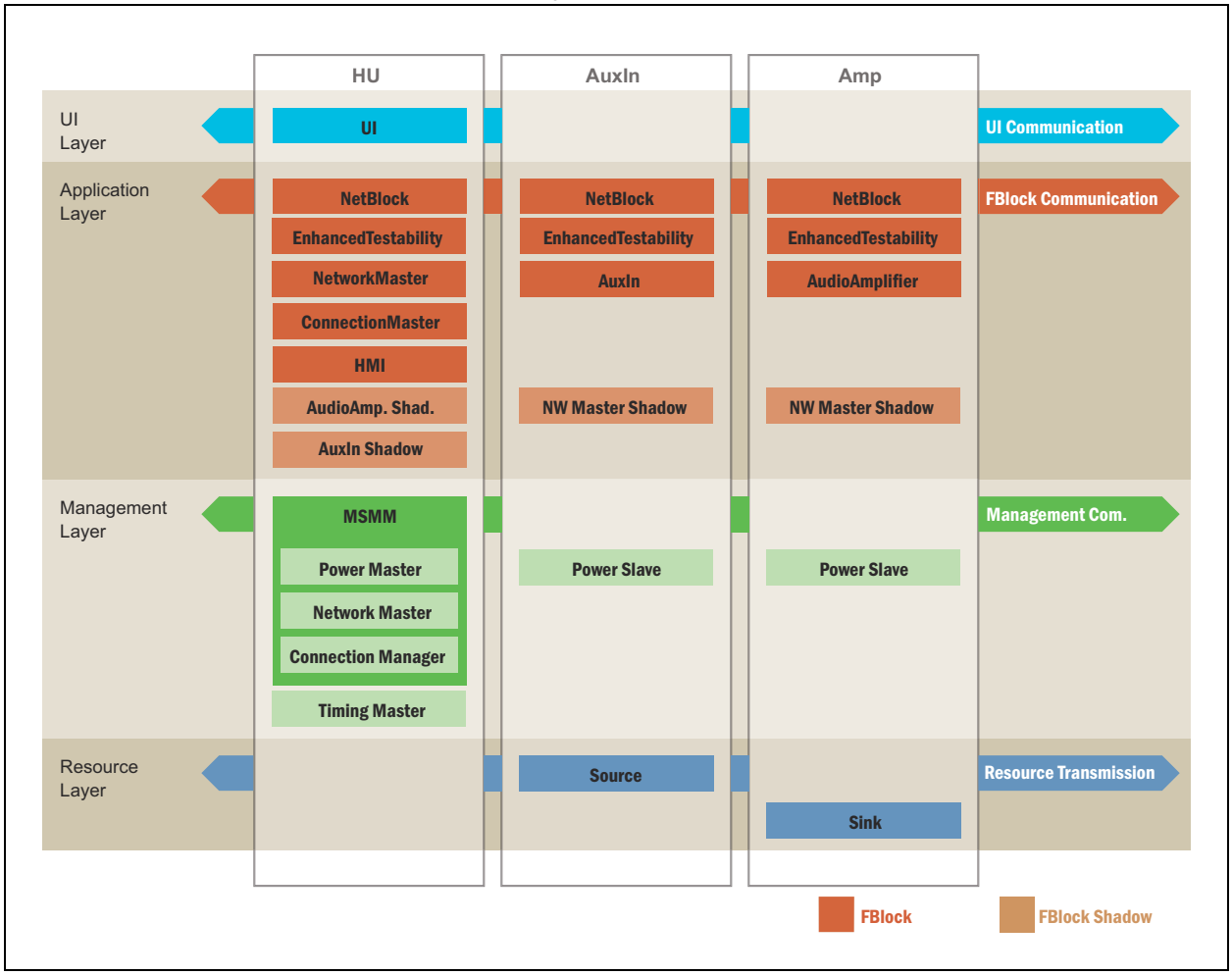
TABLE 2-1: MOST ToGo BASIC SYSTEM CATALOG

FBlock \ Device →	Master	Source	Sink	Notes
NetBlock	X	X	X	Provided by NetServices
EnhancedTestability	X	X	X	Provided by NetServices
AuxIn		X		User
AudioAmp			X	User
NetworkMaster	X			Provided by MSMM
ConnectionMaster	X			Provided by MSMM
HMI	X			User
Shadow FBlocks				
NetworkMaster Shadow		X	X	Provided by NetServices
AuxIn Shadow	X			User
AudioAmp Shadow	X			User

Now that the functionality of the system has been defined and partitioned across the network, the block diagram of the *MOST ToGo* basic system can be shown as a functional diagram as seen in Figure 2-2.

© 2014 Microchip Technology Inc.

FIGURE 2-2: MOST ToGo BASIC SYSTEM, FUNCTIONAL DIAGRAM

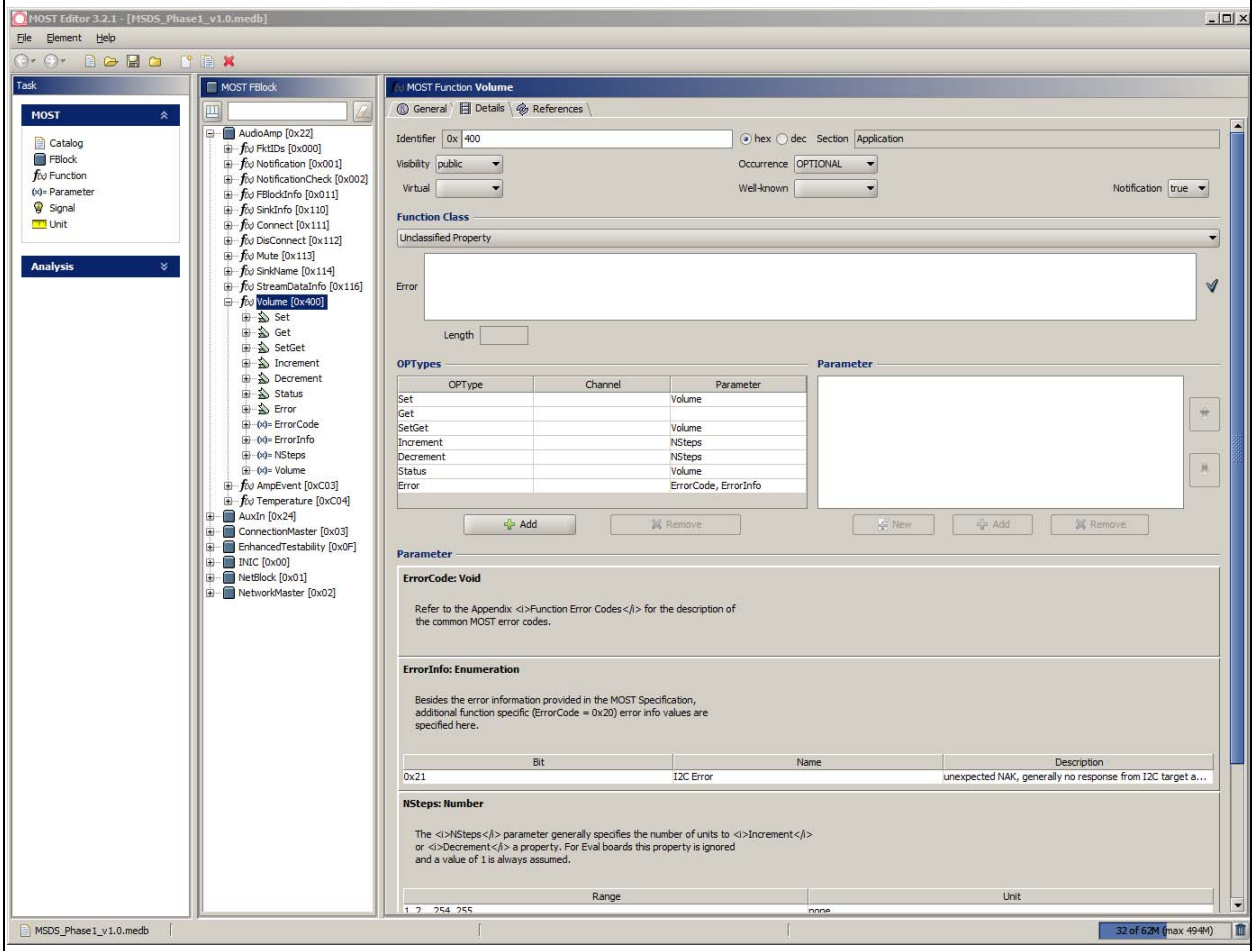


2.6 FUNCTION CATALOGS

This section describes how the FBlocks are defined and how they are combined into a Function Catalog that forms the definition of the system. As mentioned previously, the MOST Cooperation maintains a set of FBlock definitions that are meant to be used as a starting point. There are specific function blocks defined for common functions like `AudioAmp`, `AuxIn`, `PhoneBook`, and `AmFmTuner` for example, in all there are about 20 FBlocks completely defined. There is also a General FBlock to be used as a basis for creating new custom FBlocks, think of it as base class. Even the fully defined FBlocks like `AudioAmp` have been derived from the General FBlock.

The FBlocks themselves are defined in an XML file whose elements are specified by an XML schema (Fcat.xsd). Earlier versions used an XML DTD (Document Type Definition) (Fcat.dtd). The MOST Cooperation provides a tool, the MOST Editor (see the MOST Editor User Manual 3V2 [4]), which is used to import, edit, or create new FBlocks. The MOST Editor and FBlock XML files can be downloaded by MOST Cooperation members from the web site: (www.mostcooperation.com).

FIGURE 2-3: SCREEN SHOT OF MOST EDITOR



The MOST Editor is used to create the full hierarchy of elements to create a function catalog:

- Create a Function Catalog which is a collection of FBlocks
- Create and define the FBlocks which are a collection of functions (Methods and Properties)
- Create and define the Methods and Properties of each function
- Create and define the parameters of each Method or Property

The screen-shot above shows the *Volume* property of FBlock `AudioAmp`. It can be seen that the FBlockID for AudioAmp is defined to be 0x22, the function ID (FktID) for *Volume* is 0x400, and that the Volume property supports 7 OpTypes (Set, SetGet, Get, Increment, Decrement, Status, and Error). The first 5 OpTypes are used by controllers (clients) to send messages (commands) to FBlock `AudioAmp`, while the Status and Error OpTypes are responses from the FBlock back to the controller – these are the messages the shadow FBlock in the controller receives.

Each element of the Function Catalog has its own unique Version number. The Function Catalog itself has an overall Version that identifies the whole catalog. Each FBlock has a Version, and each function in the FBlock has its own version.

© 2014 Microchip Technology Inc.

When updates of a Function Catalog are received, the versions of each element can be inspected to see what has changed.

While defining the elements of the Function Catalog, the designer can include documentation that becomes part of the XML. Each parameter of each function can get a full description of its intended use and any exceptions or special use cases can be described. The MOST Editor includes a tool that can generate an HTML file from the XML catalog. This file can be viewed in any browser and serves as the full documentation of an FBlock. The pdf files that are provided by the MOST Cooperation of existing FBlocks are created from this HTML output after some post processing with a word processor to add some formatting and header and footer information. See Figure 2-4.

FIGURE 2-4: EXAMPLE OF HTML OUTPUT SHOWING DOCUMENTATION OF THE FBlock AudioAmp BASS FUNCTION

Bass (0x202)

Occurrence: Conditional

Provides a Bass Boost function in 2dB steps from 0dB to +24dB. Therefore there are 12 steps, 0 is flat and 12 (0x0C) is +24dB

Format of Function

Function classes: Unclassified Property

FBlock	Function	OPType	Parameter
AudioAmp (0x22)	Bass (0x202)	Set	Bass
		Get	-
		SetGet	Bass
		Increment	NSteps
		Decrement	NSteps
		Status	Bass
		Error	ErrorCode, ErrorInfo

Parameter

Bass

Bass setting of the Bass boost property. 0 = Flat, 12 = +24dB. Each step provides a +2dB boost.

Basis data type	Exp.	Range of values	Step	Unit
Unsigned Byte	0	0x0...0xC	1	none

NSteps

The *NSteps* parameter generally specifies the number of units to *Increment* or *Decrement* a property. For Eval boards this property is ignored and a value of 1 is always assumed.

Basis data type	Exp.	Range of values	Step	Unit
Unsigned Byte	0	0x1...0xFF	1	none

ErrorCode

ErrorCode: 0x01-0x0C;0x20;0x40-0x43

The XML definition files of a function catalog have uses beyond just being a file format to share specifications and generate documentation. Tools such as the OptoLyzer® MOCCA network analysis tool use the XML file to decode message on the network and generate a human readable log in the Viewer screen. The XML file can also be used to generate stub code, and this is done by the K2L Automotive Test System Tool (ATS) [5] which can be used to generate PC test code for a system. The Microchip Port Message Viewer v6+ [6] tool also uses the XML function catalog file to decode the message traffic between the EHC and the INIC.

© 2014 Microchip Technology Inc.

2.6.1 Creating the Function Catalog

Typically the OEM creates the master function catalog, which includes all functions in the entire system. From that master catalog the OEM can create a separate function catalog for each device in the system that includes only the FBlocks that the device should implement. This becomes a major part of the functional specification of a device.

To create the function catalog, the OEM should create a new, empty function catalog with the MOST Editor. Then import all of the predefined FBlocks needed for the system from the MOST Cooperation. For the basic *MOST ToGo* system these include:

- `NetBlock`
- `Enhanced Testability`
- `NetworkMaster`
- `ConnectionMaster`
- `AuxIn`
- `AudioAmp`

There is not a predefined FBlock for `HMI`, so this will be created later.

The first four FBlocks are in the administrative category and should be used without modification to start with. FBlock `NetworkMaster` and `ConnectionMaster` are implemented by MSMM, so that the user application does not have to implement them directly. `NetBlock` and `EnhancedTestability` are implemented by NetServices. So, it is the application FBlocks like `AuxIn` and `AudioAmp` that require the most work.

To create a unique version of a predefined FBlock like `AudioAmp`, the designer should:

- Import the latest version of the FBlock from the MOST Cooperation web site
- Study all of the functions to determine which ones are needed by the system
- Delete the functions that are not needed
- Delete the parameters that are only used by the deleted functions
- Update the documentation of each function and parameter
  - Predefined functions will have some rudimentary description when imported. This should be expanded and / or clarified for the intended use in this system
- Add new custom functions required by the system design
- Add new parameters required by the new functions
- Document each new function and parameter with descriptive text according to the intended use in this system
- Set the base revision level of each function and parameter – presumably these will all start at 1.0.0

To create FBlock `AudioAmp` for the *MOST ToGo* basic system for example, the imported version from the MOST Cooperation has about 50 functions defined. The final *MOST ToGo* version has only about 15. One custom function was added, *Temperature*, which is meant to indicate the temperature of a heat sink on a high power amplifier. In the MOST50 ePhy *MOST ToGo* kit, it is simply a

© 2014 Microchip Technology Inc.

temperature sensor on the board, but is used to show an example of a custom property.

Creating a custom FBlock such as `HMI` is done in a manner very similar to modifying an existing FBlock, the difference is the design starts with the General FBlock from the MOST Cooperation. The FBlock designer should:

- Import the latest General FBlock from the MOST Cooperation
- Rename it to the desired new FBlock name (HMI for example)
- Assign it a unique FBlock ID if it is not already defined by the MOST Cooperation
- Add the new custom functions required by the system design
- Add the new parameters required by the new functions
- Document each new function and parameter with descriptive text according to the intended use in this system
- Remove the General FBlock functions that are not needed by the new FBlock
- Set the base revision level of each function and parameter – presumably these will all start at 1.0.0

Once the entire catalog is built – the overall catalog should be given a revision number and a unique name, such as MTG_Basic for the *MOST ToGo* basic system. At this point the HTML documentation file can be generated and checked. There is usually some editing of the source XML function catalog file required to get a nice looking, accurate HTML output. The OEM may then split the function catalog into separate ones for each device in the network. For the *MOST ToGo* basic system, there would be one for the Master, the Sink, and the Source nodes. Alternatively, the OEM could just give the master catalog to each Tier 1 with explicit instructions on which functions should be implemented for each device. In either case, all parties will eventually need the master catalog for the tools like the OptoLyzer MOCCA network analyzer to use for debugging.

2.6.2 Simulation

Once the master function catalog is done, the whole system can be simulated with a tool from K2L™ (www.k2l.de) called the Automotive Test System (ATS). The ATS allows importing a MOST network function catalog in XML format and generates C# assemblies for the Visual Studio environment. The assemblies contain stub code for all FBlocks and shadows as well as setting up a ‘virtual MOST network’ on a PC where messages sent from a virtual device are delivered to the targeted virtual FBlock. The tool has a viewer window that shows messages on the ‘network’ that look the same as message traffic that would be seen on a real network with the same devices.

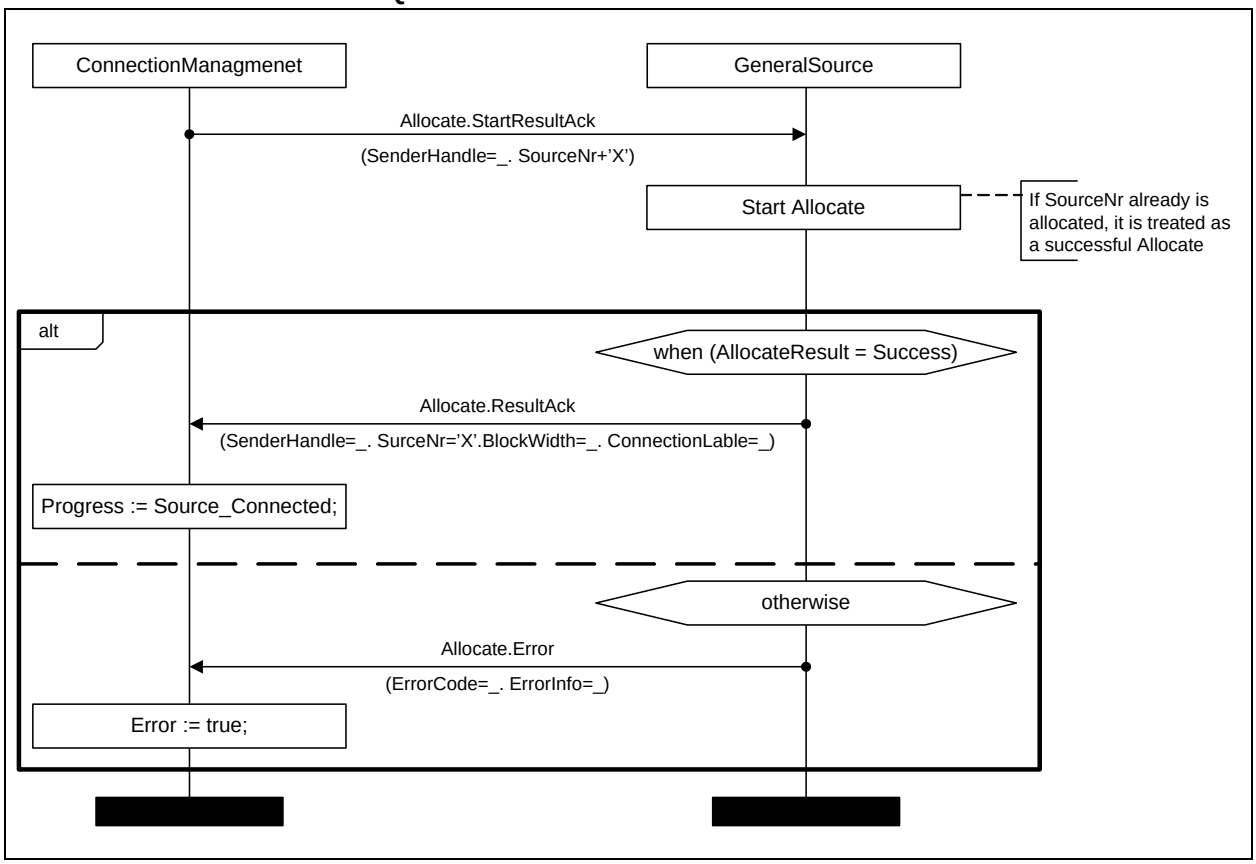
The K2L ATS tool can also be used in later stages when real hardware is available for so called ‘restbus’ simulations (simulating the ‘rest of the bus’) when only some hardware is available. This is useful to a Tier 1 which has built a single device and needs to test it as if it is in the final network, but doesn’t have access to the other devices yet. ATS is also useful for test automation like end of line testing when testing a single device, similar to unit testing code.

© 2014 Microchip Technology Inc.

2.7 DYNAMIC SPECIFICATION

The function catalog captures all of the functional specification of the system. However it says nothing about the timing or order of events. This domain is covered by the MOST Dynamic Specification [7]. This sequence information is given in the Dynamic Specification in to the form of Message Sequence Charts (MSCs). Below is an example of the sequence required for *Allocate*, the process a source uses to puts its streaming data on the network.

FIGURE 2-5: MESSAGE SEQUENCE CHART FOR ALLOCATE.



The MOST Dynamic Specification [7] has sections for:

- Network Master – handled by FBlock `NetworkMaster` of MSMM
- Network Slaves – handled by FBlock `NetBlock` provided by NetServices
- Connection Management – handled by FBlock `ConnectionMaster` of MSMM
- Power Management – must be handled by the application
- General Source and Sink management – must be handled by the application

The most complex part of the MOST Dynamic Specification [7] by far is the behavior of the Network Master, the Network Slave's responses, and Connection Management (about 70% of the specification). Luckily for developer's this is all handled by MSMM and NetServices. The developer must keep in mind when implementing the source and sink functions (`AuxIn` and `AudioAmp`) that the design must meet the Dynamic Specification as well as the Functional Specification given by the function catalog.

© 2014 Microchip Technology Inc.

For the *MOST ToGo* basic system, the MOST Dynamic Specification [7] covers all of the functionality used in the system. That is, no further dynamic specifications need to be developed for any of the functions used by the basic system. If the system had some other device like an FBlock `AmFmTuner` then that function might need an additional dynamic specification since nothing like that is covered in the MOST Dynamic Specification [7].

2.8 POWER MANAGEMENT

Power management in a MOST network deals primarily with system wakeup, going from a low power sleep mode (*sleep power state*) into normal active state (*active power state*), and system shutdown, returning back to the *sleep power state*. The MOST Specification 3.0 [2] states that one device should be the Power Master and be in charge of power management for the entire network. Because this function is interrelated with the Network Master, both of these functions should be on the same device.

There is no FBlock for the Power Master, as it does most of its work when the network is off, either before the network is started, or after the network has shut down and there is no communication between nodes. The MOST Specification 3.0 [2] and the MOST Dynamic Specification [7] both have sections on power management that describe how the devices are to behave.

Also, the INIC Hardware Concepts [8] document deals extensively with power management implementation in a device from a hardware standpoint. The MOST ToGo Hardware Principles [9] document describes the power management as implemented on the evaluation boards used in the *MOST ToGo* basic system.

Typically the Power Master device wakes up due to some local event such as a button press (power button) or a CAN message that is triggered from a door open event or key insertion etc. The power master must determine if the wakeup event is valid, then it will wakeup the rest of the network. There are two ways the power master can wake the network; by activity – simply sending out the modulated signal on the network; or by using an electrical control line (ECL).

The ECL is the preferred method as using activity requires that each device have a low power activity detector which is active while the device is in *sleep power state*. This is no problem for optical or coaxial networks as this activity detection is part of the Fiber Optic Transceiver (FOT) or Coax Transceiver. For electrical networks (ePhy) as used in MOST50 there is no FOT, the network signal goes directly to the INIC via some passive components, and the INIC is not powered while in *sleep power state*.

There is a solution for ePhy networks in the Microchip MPM85000 Power Management device for MOST networks. The MPM85000 (MPM85000 Automotive Power Management Device Data Sheet [10]) has a low power activity detector for ePhy networks and is able to wake a node based on network activity. Even with an activity detector, the ECL is still the preferred method as it wakes up the network faster, and the ECL provides another channel for diagnostics. The MOST Electrical Control Line Specification [11] defines the properties of the ECL and a protocol for diagnostics using only the ECL. Electrically the ECL is like a LIN signal, nominally at 12V, but devices can pull the line low for signaling. See the MOST Electrical Control Line Specification [11], the INIC Hardware Concepts [8], and the MOST ToGo Hardware Principles [9] for a complete description of the ECL usage.

The Power Master is also in charge of actually starting the network via a command to NetServices and checking to see that the network actually starts up and locks. If the network does not lock, then the Power Master is responsible for

© 2014 Microchip Technology Inc.

doing network startup retries. If the network cannot startup, the Power Master will initiate system diagnostics via the ECL. These diagnostics are defined in the MOST Electrical Control Line Specification [11] and are called System Tests.

In the *MOST ToGo* basic system, the Network Master device is also the Power Master. An application module called PMGT (pmgt.c) implements most of the functionality required of the power master. The ECL is monitored and controlled by a state machine, the ECL Software Module (eclsm.c) that can detect and generate wakeup pulses on the ECL, and can generate or participate in the ECL System Tests. The ECLSM is always active while the node is powered up.

An add-on module to MSMM is being prepared that will implement the ECLSM. But, for the *MOST ToGo* basic system, the ECLSM is part of the application. The PMGT module interfaces to the ECLSM and together they form the Power Master function of the *MOST ToGo* basic system. More details on the PMGT module and the ECLSM are in the implementation sections.

2.9 FEATURE CREEP

As with most projects, once the design is finalized and schedules created, someone decides that some features should be added (usually without changing the schedule or budget). The *MOST ToGo* project is no exception, as it was decided to add more functionality to the master device. It was decided to add both a sink and a source to the master device by implementing FBlock `AuxIn` and FBlock `AudioAmp`. Since the same type board is used for each node, the hardware is already there to support both. This added functionality will give the *MOST ToGo* basic system two sources and two sinks which will lead to more interesting demos.

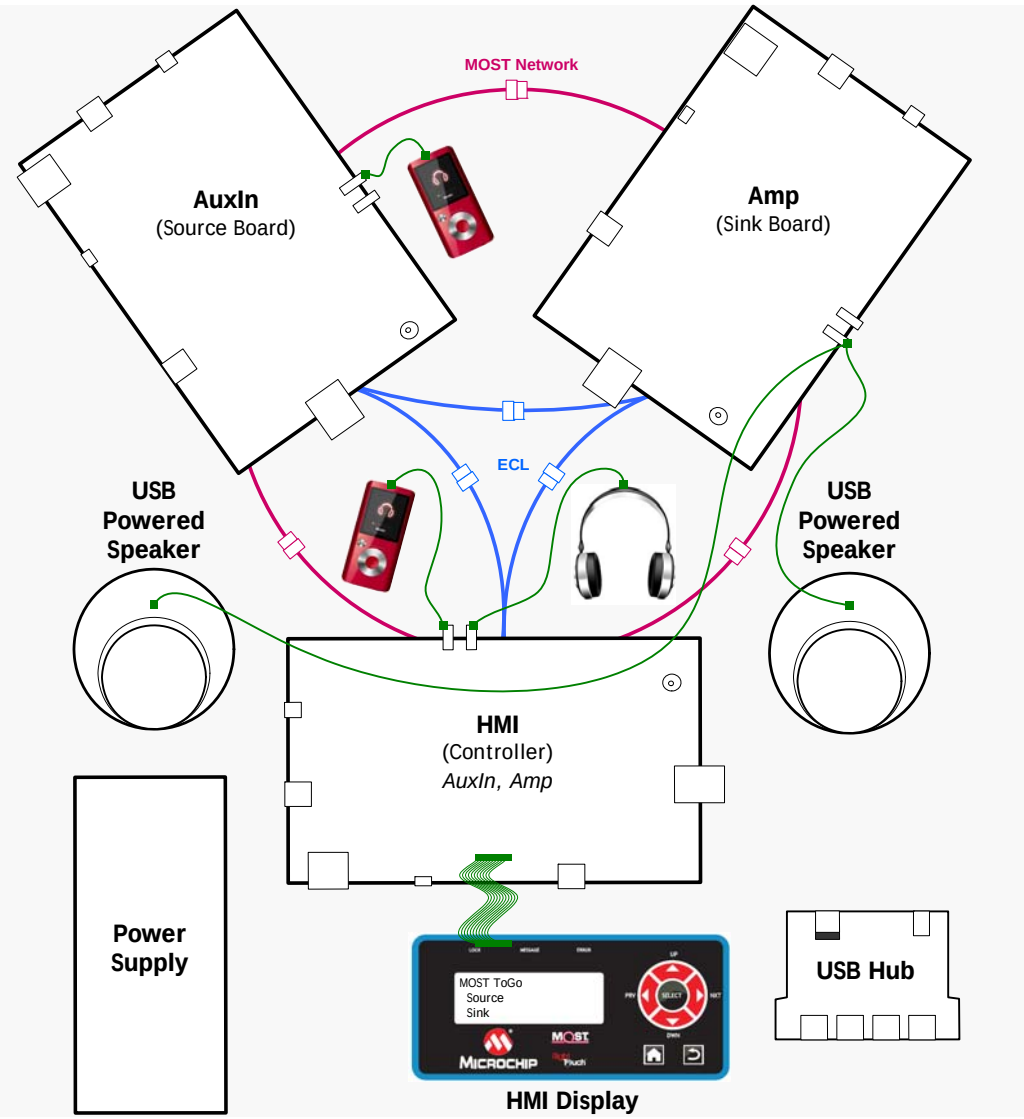
Adding this functionality will show several things about a MOST system:

- An FBlock and its FBlock shadow can exist in the same device.
- The FBlock implementation is identical regardless if it resides on a master or slave device.
- A device may contain several unrelated functions and implement FBlocks that do not interact with each other (e.g. FBlock `HMI` and FBlock `AudioAmp` do not interact).
- Shows that a controller (AV_Manager in this case) does not need to know where the FBlocks are located in the network, and it works the same whether the FBlock is on the same device are located across the network.
- Since the added FBlocks are duplicates of those that already exist, there is no need to modify the function catalog.

Figure 2-6 below shows the modified block diagram, adding the new features and the ECL signal.

© 2014 Microchip Technology Inc.

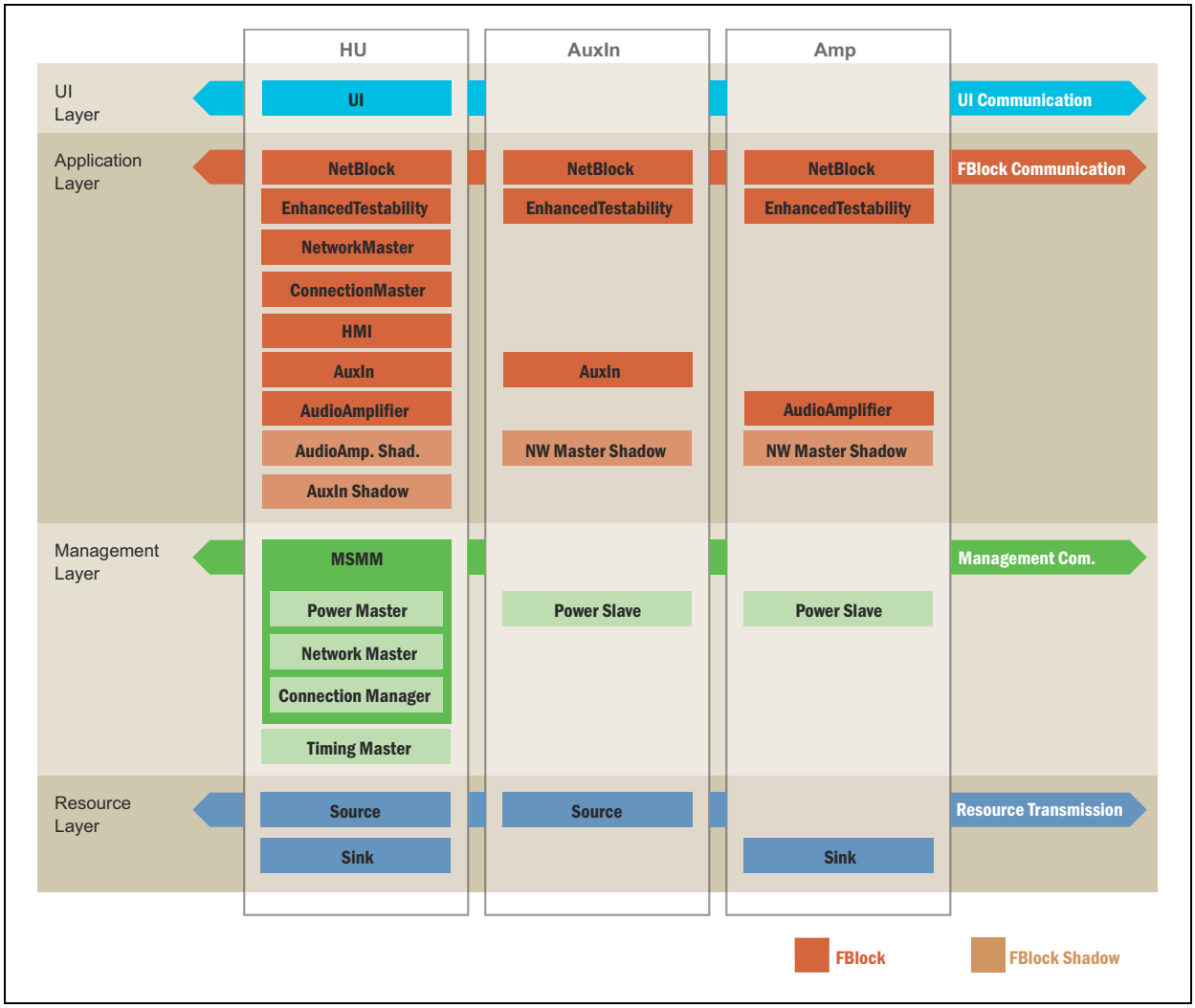
FIGURE 2-6: MOST ToGo BASIC SYSTEM KIT



The power supply is a 120/240Vac to 12V supply that powers all of the boards. The USB hub connects to the USB debug print port of each board and powers the speakers. A single USB connection back to a PC allows the user to see the debug output of all boards with a terminal program like TeraTerm or using the PortMessageViewer application that can also decode all the messages.

Adding the FBlocks `AuxIn` and `AudioAmp` to the master node has also altered the functional block diagram of the system. The revised diagram is shown in Figure 2-7 below.

FIGURE 2-7: MOST ToGo BASIC SYSTEM FUNCTIONAL BLOCK DIAGRAM



2.10 SUMMARY

As can be seen, the specification of a MOST system consists of many parts, of which all except the actual hardware itself are covered by the MOST Specification 3.0 [2]. To define a system the OEM must define all of the elements described above:

- Hardware Specification – electrical properties of the device including the application circuitry, power supply, INIC device and speed grade (MOST50, MOST150 etc), and whether the network physical layer is optical (oPhy), electrical twisted pair (ePhy), or coax (cPhy).
- Functional Specification – in the form of a MOST Network Function Catalog
- Dynamic Specification – from the MOST Dynamic Specification [7] plus any addition Message Sequence Charts necessary
- Power Management Strategy – Wakeup sources, wakeup strategy (activity, ECL, or both), retry strategy, and diagnostics.

© 2014 Microchip Technology Inc.

Chapter 3. Device Implementation

Once the specification of the system is done, the implementation of the devices can begin. The devices are first and foremost embedded systems that implement some application specific functions such as an amplifier or tuner. They also include the hardware needed for a MOST network which consists primarily of the INIC and the physical interface (oPhy, ePhy, or cPhy). The majority of the hardware work is the application circuitry itself, the network component is actually fairly small. Care must be taken to make sure the power supply circuitry works with the MOST network power management scheme as described in the INIC Hardware Concepts document. It may involve adding an MPM85000 power management device which will add the hardware needed for ECL, ePhy activity detection for MOST50, and wakeup event qualification.

The firmware implementation is similar to the hardware implementation in that a major portion of the work is just a standard embedded system programmed to implement the device's functionality. The network component is added on and interfaced to the normal embedded firmware. The firmware can be written without the network component at all, and the hardware can be tested for functionality without the network. The device can be connected to a network test tool like the OptoLyzer MOCCA operating as a network master and the network should lock without any EHC code at all. The main.c file in the *MOST ToGo* code has a sample INIC test routine that will test the connection between the EHC and INIC without needing NetServices. So, the device itself can largely be coded and tested without having to integrate NetServices, MSMM, or implementing any FBlocks.

In the sections below the primary focus is on adding the network component to an application. For this reason the *MOST ToGo* basic system and application has been made as simple as possible (a single source, or single sink) so the focus can be on the network implementation without a lot of complex code dedicated to the application logic. The emphasis is on how to integrate NetServices and MSMM into an embedded application, how to define and implement the FBlocks in code, and how to interface the NetServices command interpreter to the application logic.

3.1 MOST ToGo BASIC SYSTEM HARDWARE

The *MOST ToGo* basic system uses three OS81092 ePhy Evaluation Boards designed for the *MOST ToGo* project. The board has several features that are not needed for the basic system, but are meant for future expansions to the project. A basic block diagram of the MOST50 board can be seen in Figure 3-1. The OS81092 ePhy Evaluation Board for the *MOST ToGo* system has the following major features:

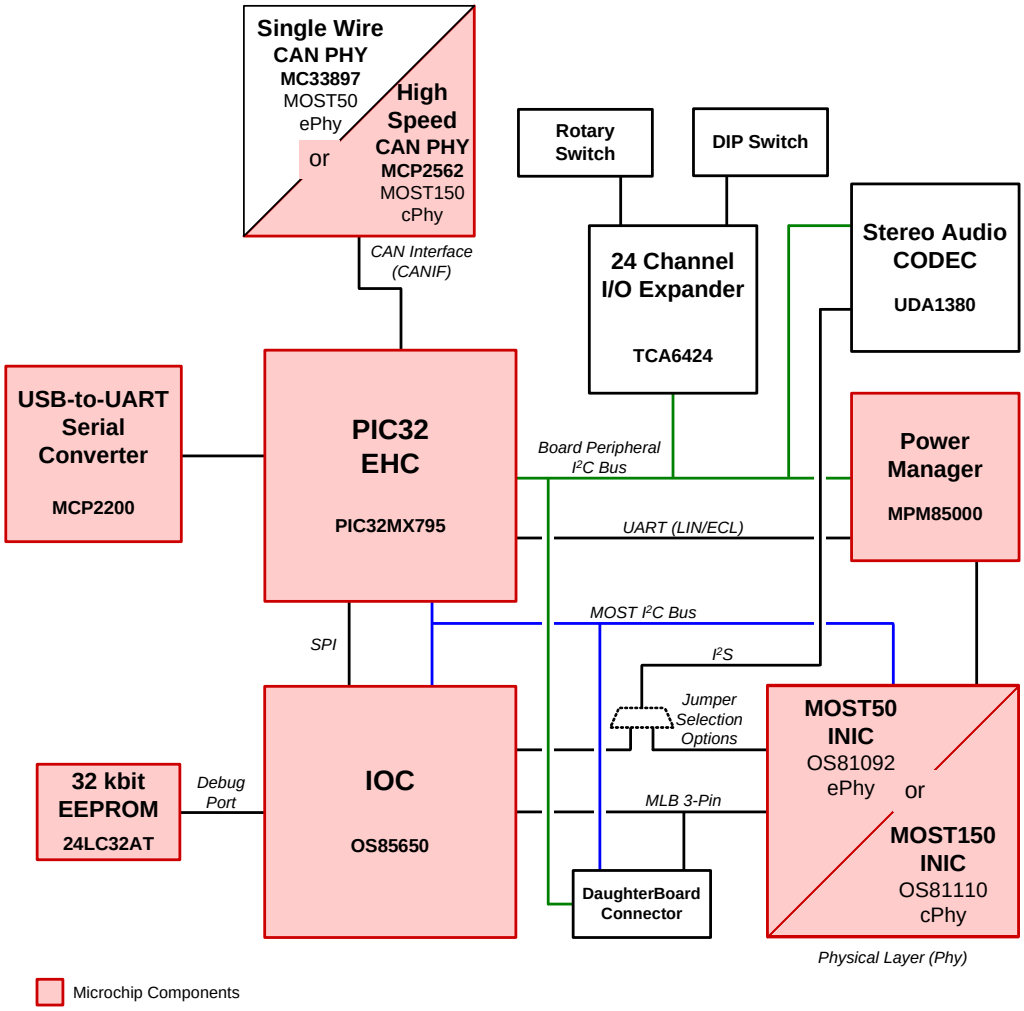
- OS81092 MOST50 INIC interfaced to the EHC via I^2C
- OS85650 I/O Companion device interfaced to the EHC via I^2C and SPI
  - The IOC can provide additional streaming ports, DTCP encryption / decryption, and a high speed interface to the INIC via the MediaLB bus. It is not used in the basic system
- MPM85000 power management device, interfaced to the EHC via I^2C
- GPIO port expander providing additional I/O pins, interfaced to the EHC via I^2C
  - Supports DIP switch (application options), a rotary switch and status LEDs

© 2014 Microchip Technology Inc.

- A high speed expansion port for daughter cards with MediaLB®, I²C, and power
- A low speed expansion port (not shown) with I²C, RS-232, and power
  - This port is used for the HMI display (basic display)
- MPC2200 USB to UART converter for debug print output
- MC33897 CAN interface – future features showing CAN wakeup and CAN gateway to MOST networks

There are a lot of I²C devices, but they are split between two busses, one for the INIC and the IOC, and another for all the other peripherals. The goal is to provide the highest performance possible via I²C to the INIC as all network messages and INIC control messages (commands to INIC itself) go through this link.

FIGURE 3-1: MOST ToGo EVALUATION BOARD BASIC BLOCK DIAGRAM



Note: The block diagram above is identical for both the MOST50 ePhy Evaluation Board and the MOST150 cPhy Evaluation Board, with the exception of the Physical Layer (Phy) interface and the CAN Interface (CANIF)

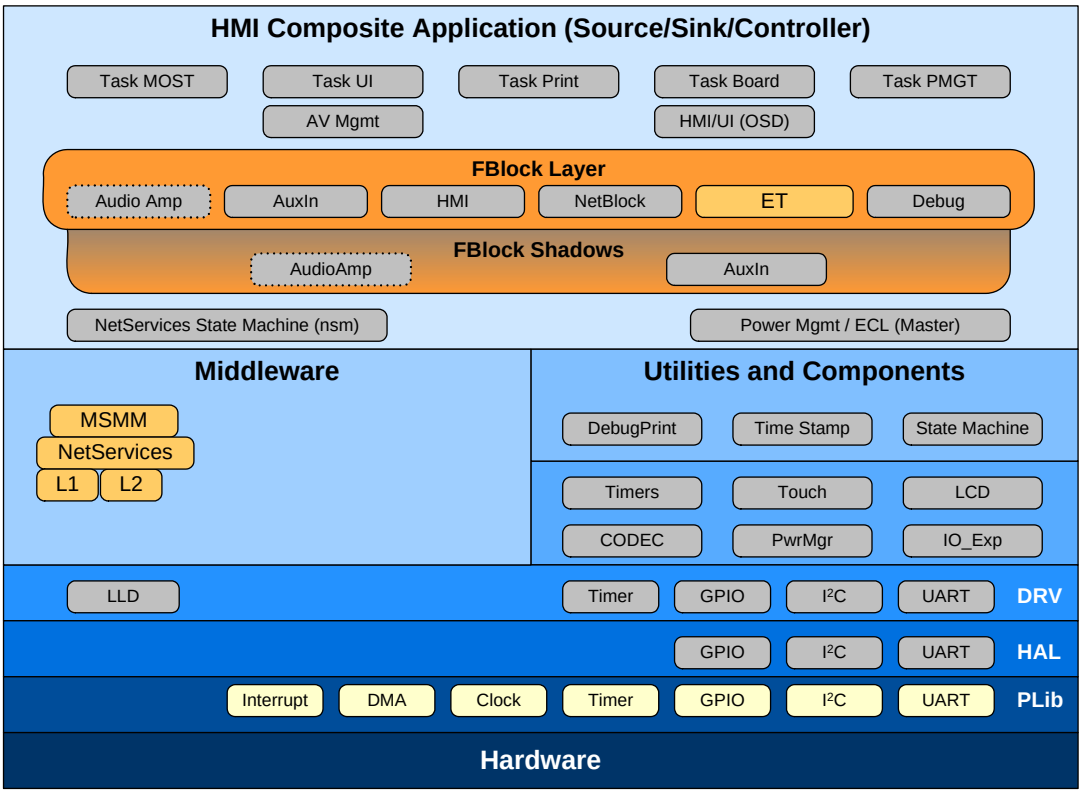
© 2014 Microchip Technology Inc.

3.2 APPLICATION FIRMWARE

As stated earlier, these devices are embedded systems. The first order of business is to create all the low level drivers for access to all the peripherals on board, including those of the EHC itself. Most microprocessor vendors have libraries of drivers for the on chip peripherals. For the PIC32, those libraries are called PLibs and are built into the XC32 compiler. The structure of the *MOST ToGo* application code is built in a layered fashion from the driver layer up to the application layer as shown in Figure 3-2 below.

The layer diagram for the HMI application is shown because the source and sink applications are subsets of the master application. The slave devices do not include the MSMM layer, the AVManagement module, the HMI/UI module, or the FBlock shadows for `AudioAmp` and `AuxIn`. The slaves do include the `NetworkMaster` shadow, and the Power Management / ECL module, but compiled for a slave device.

FIGURE 3-2: MOST ToGo STRUCTURED CODE LAYERS



Most of the lower level layers are standard embedded system code. These layers are highly hardware and application dependent. Most EHC vendors supply evaluation boards that include board support packages (BSPs) that include a lot of this code. If the application board is modeled after a vendor's evaluation board, this BSP code can be used as a starting point. Only a brief description of the lower level code for the *MOST ToGo* system is given. The complete documentation for all the *MOST ToGo* code is given in the Doxygen generated MOST ToGo Code Documentation [12] which is provided in electronic form as a compressed help file (.chm).

3.2.1 Hardware Abstraction Layer (HAL)

The HAL layer is designed to provide a barrier between the peripheral libraries that interface to the hardware, and the driver layer which the application uses. That is, the HAL layer translates between the driver layer and the peripheral libraries for any particular EHC. In the *MOST ToGo* system, it adapts the driver layer to the PLibs. The goal is to keep the driver layer the same regardless of the underlying processor. This separation facilitates porting of the application code to different processors.

3.2.2 Driver Layer

This layer consists of drivers for on chip resources such as UARTs, GPIO, etc. and also for on board peripherals such as CODEC, I/O expander, LCD etc. The IO Expander driver and the GPIO driver work together to give a GPIO system that has a common interface for GPIO pins on the EHC itself or on the port expander. Below is a list of the drivers included, see the *MOST ToGo* Code Reference help file for full details.

- `drv_clk` – setup the PIC32 PLL to generate proper CPU clock frequencies
- `drv_codec` – Routines to initialize the CODEC and set the Volume, Bass, Treble, Mute etc. of the CODEC. These routines are used by both AuxIn FBlock and AudioAmp FBlock.
- `drv_gpio` – routines to interface to the GPIO pins of the CPU
- `drv_i2c` – interface routines for communicating with devices on any of the I^2C busses. This driver is especially critical since the I^2C bus is the main communication line between the EHC and INIC.
- `drv_int` – Interface routines to the interrupt logic of the CPU
- `drv_ioexp` – driver for the on board I/O Expander, which expands the number of GPIO pins available. Designed to make accessing these pins look just like accessing GPIO pins on the EHC.
- `drv_lcd` – Interface routines to the LCD display which is connected via the low speed expansion port to an EHC UART
- `drv_pm` – low level interface to the MPM85000 power manager chip which is connected via I^2C
- `drv_timer` – interface to the system tick hardware timer and software timers
- `drv_touch` – interface to the RighTouch™ capacitive sensor which provides the buttons and LEDs on the user interface. This device is connected via I^2C to the EHC via the low speed expansion port.
- `drv_usart` – Interface routines for any of the EHC's UARTs

The I^2C driver has some special functionality to aid in debugging. An ASCII representation of every I^2C transaction is also sent to the debug printing routines described in the utility section below. Example 3-1 Sample Line for a Message Being Sent to INIC shows a sample line for a message being sent to INIC.

EXAMPLE 3-1: SAMPLE LINE FOR A MESSAGE BEING SENT TO INIC

```
000:00:00:884> s40.00.0C.05.04.00.01.01.00.00.30.00.02.01.41p
```

At the beginning of the line is a time stamp from the system clock provided by `drv_timer`. The `s40` indicates an I^2C start condition followed by the I^2C address of 0x40. The data sent to that address follows up until the last byte which

© 2014 Microchip Technology Inc.

is followed by a ‘p’ indicating the stop condition. The Port Message Viewer v6+ [6] program from Microchip can interpret these messages via the function catalog to show them in an FBlock message format.

3.2.3 Utilities

There are several utilities implemented that provide some general services to the application, like printing. The utility routines are listed below.

- `fprint` – encapsulates the standard C library fprint services into some macros that enable printing to be enabled or disabled at runtime.
  - `TRACE` – a print routine meant for standard messages from the application indicating program flow. It can be disabled with DIP Switch 2
  - `TRACE_T` – same as above, but prints a time stamp along with the other print text
  - `TRACE_DEBUG` – a print routine meant for debug output rather than announcing normal operation. It can be disabled with DIP Switch 3
  - `TRACE_T_DEBUG` – same as above, but includes a time stamp along with the message
- `bip_buffer` – a special type of ring buffer used to buffer print data so it can be printed in the background.
- `state_machine` – handles execution of the several state machines implemented in the application. Takes events targeted to a state machine, performs the specified action based on the event and current state, and sets the next state.
- `time_stamp` – provides a text string based on the current system time
- `util_cvt` – provides various conversions between bytes, words, longs.

3.2.3.1 DEBUG PRINTING

The application makes extensive use of debug printing, like the I^2C traffic shown above. It is also used to print the internal actions of NetServices by using the NetServices Trace module. The upside of all the printing is that it makes it very easy to trace the actions of the application code. Most debugging can be done just by looking at the output, a hardware debugger such as a PICKit3 is seldom needed. There are a couple of downsides though. First, all the strings take up a lot of code space, second, printing through a UART takes a lot of application time.

The *MOST ToGo* application has the ability to do all of the printing in the background. If enabled, the print routines send the output to a ring buffer (`bip_buffer` utility) and a task called in the main loop checks to see if the UART buffer is empty, if so, then the UART buffer (8 characters on a PIC32) is quickly filled from the ring buffer. The characters are then sent out while the application continues to run. Background printing speeds up the application significantly (from reset until the network has been fully setup goes from about 700 ms to 200 ms). The downside of background printing is that the printing lags behind what the application is doing. If the application gets stuck, the last printed output may have nothing to do with where the application is getting stuck. The background printing can be turned off so that the output happens in real time.

The debug print routines are defined by macros and these can be disabled so that none of the printing routines or strings get compiled. This will release all of the code space being used by the strings and speed up the application. These routines would normally be disabled in a release version of the application.

© 2014 Microchip Technology Inc.

3.2.4 Board Support Package

The specifics of how GPIO pins and how peripherals such as UARTS and I^2C port are connected are provided in the *board.c* and *board.h* files which form the board support package for the evaluation boards used in the *MOST ToGo* basic system. The *board.h* file has a macro definition for every GPIO pin, the name of the macro usually matches the net name of the signal on the circuit board. See Example 3-2 GPIO Pin Definition:

EXAMPLE 3-2: GPIO PIN DEFINITION

```
/*! EHC Poweroff GPIO number Pin L7 PORTB RB13 */
#define GPIO_NUM_EHC_PWROFF                              (29)
```

Then in the *board.c* file, there are 'helper' functions for each GPIO pin that give the application routines to call to manipulate specific pins. See Example 3-3 GPIO Configuration and Helper Functions

EXAMPLE 3-3: GPIO CONFIGURATION AND HELPER FUNCTIONS

```
/*! EHC Poweroff GPIO number 29 */
drv_gpio_pin_t gpio_ehc_pwroff =
{
        .number = GPIO_NUM_EHC_PWROFF,
        .flags = DRVGPIO_PF_DIR_OUTPUT_BF() |
    DRVGPIO_PF_ODEF_HIGH_BF() |
    DRVGPIO_PF_OEN_TRUE_BF() |
    DRVGPIO_PF_OTYPE_OPEND_BF(),
};
void pwroff_uc_assert(void)
{
  drv_gpio_pin_clear(&gpio_ehc_pwroff);
}
void pwroff_uc_deassert(void)
{
  drv_gpio_pin_set(&gpio_ehc_pwroff);
}
```

The example above is for the EHC power off pin that goes to the MPM85000 power manager device. The EHC can pull this pin low to keep the board from shutting down. The pin structure has the GPIO pin number that was defined in the *board.h* file, as well as the flags that determine how the pin should be configured. There are additional elements to the structure if the pin is an interrupt pin, but that doesn't apply here. An initialization routine (`board_init()`) initializes each GPIO pin according to how it is defined in the pin structure. Then the application simply uses the `pwr_uc_assert()` and `pwr_off_uc_deassert()` functions to change the pin. Note that the application does not even have to know that asserting the signal pulls it low, it just has to know that it wants to assert the signal.

The *board.c* file is also where the UARTs are assigned to their particular function (e.g. debug print, or LCD) and the baud rates specified. Also this is where the I^2C addresses of all of the peripherals on the I^2C bus are assigned.

© 2014 Microchip Technology Inc.

3.3 INTEGRATING NetServices

MOST NetServices is delivered as a set of C source files that the user includes in the build of the project. It is divided into two major layers fittingly called Layer 1 and Layer 2. There are some add on layers such as MOST High Protocol, but the *MOST ToGo* basic application only needs Layer 1 and Layer 2. Layer 1 provides the low level interface to INIC and includes the MOST network messaging interface which sends and receives network messages and packet data, and handles buffer management. Layer 2 provides the command interpreter, the notification service, and the address handler. The address handler is what frees the application from having to know the physical address of the devices it wants to communicate with.

Each layer of NetServices has its own user manual that describes the API of each function and gives the implementation details. See the MOST NetServices Layer I User's Guide [13] and MOST NetServices Layer II User's Guide [14].

3.3.1 Adding Layer I

MOST NetServices Layer I is comprised of several source files, each of which add a specific functionality to NetServices. The major services and the files that implement them are:

- NetServices Kernel – *mns.c* – the core of Layer I, NetServices internal operating system
- Port Message Service – *pms.c* – Port messages refer to the protocol between the EHC and INIC. There are transmit and receive components. The transmit side gets messages in the form of a C structure and encodes that data into a port message to send to INIC. On the receive side, the PMS service gets port messages from INIC and decodes them into a C structure to pass on to NetServices.
- Message Buffer Management – *mbm.c* – The user statically allocates a pool of memory for NetServices to use. The MBM service manages that pool of memory, assigning segments (buffers) to the application and internal uses as needed.
- Virtual MOST Supervisor –*vmsv.c* – this is essentially a shadow for all of the status messages coming from INIC, i.e. this is how NetServices keeps track of the state of INIC.
- Application Message Service – *ams.c* – this service handles the MOST network control messages going to/from the FBlocks.
- Asynchronous Data Service –*wads.c* – this service handles the sending and receiving of asynchronous data packets (packet data).
- Socket Connection Manager – *wscm.c* – this service handles the configuration and control of the synchronous data channels (audio / video data).

Adding NetServices Layer I to an application involves the following steps, each of which will be explained in more detail below.

- Add all of the C source files in the Layer 1 folder to the project
- Edit the Layer I adjust file *adjust1.h* selecting the options needed by the application
- Add the NetServices include file *mostns.h* to any modules that call NetServices
- Setup the NetServices initialization structure and call the initialization function

© 2014 Microchip Technology Inc.

- Call NetServices periodically or when needed as indicated by a callback flag
- Monitor INIC for incoming messages and send those to NetServices

3.3.1.1 LAYER I SOURCE FILES

There are 13 ‘C’ source files in Layer I, and they should all be added to the project. Exactly how that is done depends on the development environment. In MPLABX which is used for the *MOST ToGo* project this is done by right clicking on the project tree at the desired location and selecting “Add existing item”. MPLABX supports virtual folders in the project tree (the folder structure of the project tree is not related to the folder structure of the files on disk). In the *MOST ToGo* project, these files are in /Source_Files/src/common/ns/L1.

There are more than 50 header files (.h files) in Layer 1. In MPLABX these do not need to be explicitly added to the project. If a ‘C’ file includes them, then the compiler will find them if they are in the same folder as the ‘C’ source files (and they are all in the Layer 1 folder). They *can* be added to the project so they show up in the virtual folder tree where they can easily be opened if needed. The source and header files of NetServices do not need to be modified, and they should NOT be modified, so there is no reason the h files need to be added to the project.

3.3.1.2 LAYER I ADJUST FILE

NetServices is highly configurable to match the application. There are many services and options that can be turned on or off. These options are selected by the user in the ‘adjust’ files for each layer. The Layer I adjust file *adjust1.h* is located in a sub-folder inside the L1 folder, *L1/UserAdjustable*. The adjust files will need to be edited to configure NetServices, and it makes sense to add these files to the project tree.

Chapter 4 of the NetServices Layer I User Manual describes each setting in the *adjust1.h* file. The *adjust1.h* file is organized in sections that roughly correspond to the major services of Layer I outlined above. Likewise, Chapter 4 is organized by these same sections, outlining the options for each service.

The complete settings of the *adjust1.h* file for the *MOST ToGo* basic application can be seen in the *MOST ToGo* Code Documentation help file. A few of the major settings are given below.

- MOST50 INIC is selected (MOST_INIC_ID), NetServices needs to know which INIC it is controlling
- NetServices Layer II and services used there are enabled, Layer I needs to know if Layer II is on top
- MOST High Protocol is NOT included, packet data is not used in the Basic application
- Memory pool is defined for MBM, a total of 8k bytes is allocated to MBM
- The AMS service is enabled (MOST network control messaging) and the retry parameters are set
- The ADS service is NOT enabled, packet data is not used
- The socket connection manager (SCM) is enabled

3.3.1.3 INCLUDE FILE FOR NetServices

Any application module that makes use of any NetServices API functions must include those API prototypes with a *#include* directive. NetServices has a single

© 2014 Microchip Technology Inc.

include file for this purpose that will include not only any API prototypes, but all of the NetServices macros and enumerations an application might need. So a single `#include "mostns.h"` in any module calling NetServices is all that is needed. Of course, the compiler will be the final judge on whether a module needs this definition.

3.3.1.4 INITIALIZING NetServices

NetServices is initialized by the application via a single call, `InitNetServices()`. Prior to this call, the application must call `GetNetServicesConfig()`, which returns a pointer to a large structure that NetServices has filled out with default values. The application must fill in the structure, then call `InitNetServices()`.

When the application needs NetServices to do something, it simply uses the API to call it. When NetServices needs something from the application (such as current system time), or to update the application with new status information (such as network state), it uses callbacks. These callbacks are setup in the configuration structure that `GetNetServicesConfig()` returns. The user defines a callback function for each status call or request that NetServices may use, and then provides NetServices a pointer to that function in the configuration structure.

Chapter 5 of the NetServices Layer I User Manual describes the process of setting up the initialization structure and has references for each element of the structure. In the *MOST ToGo* application the initialization of the structure and calling `InitNetServices()` is in the *nsm.c* module. See the `nsm_init()` routine to see how it all works.

3.3.1.5 SERVICING NetServices

The application must call NetServices periodically so that it can do its work. This is done with a call to `MostService()`. Internally, NetServices keeps a queue of tasks to do. Each time `MostService()` is called it does one of the items in the queue. Managing the queue and the priorities of the tasks on it is handled by the Layer I kernel.

NetServices uses a callback, `most_service_request_fptr()` to request service from the application. The application should just set a flag (or semaphore) that will be checked at a later time (as part of the main loop for example), and if set, call `MostService()`. Note that `MostService()` MUST NOT be called directly from the callback function as this would be recursive.

In the *MOST ToGo* application the callback function is in *nsm.c* and is called `nsm_most_service_request()`. That function simply sets a flag, `mns_service_request`, to true. In the function `nsm_service()` which is called from the main loop, the flag is checked. The code to handle the request is shown in Example 3-4 Handling the MostService Semaphore:

EXAMPLE 3-4: HANDLING THE MostService SEMAPHORE

```
if ( mns_service_request )  // if required, service MOST
{
    mns_service_request = MNS_FALSE;
    MostService();
}
```

Net Services also handles some tasks based on time. If it needs to do or check something at some time in the future, it will ask the application via a callback (`next_min_timeout_fptr()`) to start a timer. When that timer expires, the application should notify NetServices with a call to `MostCheckTimers()`. Each time NetServices is called it checks on the current system time by asking the application for the time via another callback (`get_tick_count_fptr()`). NetServices expects the time to be the number of milliseconds the application has been running. The timer tick value is only 16 bits, and NetServices will keep track of overflows in the tick count value.

3.3.1.6 CONNECTING INIC TO LAYER I

There is a lot of communication required between the EHC where NetServices is running and the INIC which is doing the work of sending and receiving messages on the network. Therefore a communication channel must be established. In the *MOST ToGo* basic application, that link is an I^2C bus. For applications that don't need to send large asynchronous data packets, the I^2C channel provides enough bandwidth for the control messages. Providing this link is the job of the Low Level Driver which is described in the next section.

3.3.2 Low Level Driver

Before Layer 1 can communicate with INIC, NetServices needs a Low Level Driver (LLD). Chapter 14 of the MOST NetServices Layer I User Manual specifies what is required of the LLD, and provides a sample skeleton implementation. The sample shown is actually for a MediaLB connection, but the *MOST ToGo* basic application uses a I^2C connection. However, the overall requirements are the same for I^2C or MediaLB and the skeleton can still be used as a template. The LLD for the *MOST ToGo* application is in *nsm.c* which is located with the application source files. There are two main functions to implement: One is to receive a message from NetServices and transmit it to the INIC, the other is to receive a message from INIC and send it to NetServices.

When NetServices has a message to send to INIC, it puts that message in a NetServices buffer (an MBM Buffer). It then looks up the transmit function in the initialization array (`i2c_tx_fptr()` for I^2C) and passes the MBM buffer to the function. In the *MOST ToGo* application that function is `nsm_lld_transmit()`. There are options for the application to send some additional data such as a handle or time stamp along with the message. If that extra data is present, there will be in an additional buffer (not in the MBM buffer) which is also passed to the transmit function. If that data is there, then it must be appended to the message in the MBM buffer. The `nsm_lld_transmit()`must:

- Get a pointer to the data in the MBM buffer
- Copy the data to a local buffer
- If the additional data is present, add that to the local buffer
- Send the local buffer to the I^2C driver
- Release the MBM buffer when done
- Return the status (success or failure) to NetServices

Getting messages from INIC to send to NetServices is a little different in that NetServices has no idea when this might occur, this action is triggered by INIC. When INIC has a message, either received from another node across the network, or an internally generated message, it will set the INIC **INT** pin low. The application must monitor this pin, either by polling or with a CPU interrupt, and read the message from INIC. In the *MOST ToGo* application, the **INT** pin is polled

© 2014 Microchip Technology Inc.

by the routine `mns_receive_inic_message()` which is also in the *nsm.c* module. The NSM module is called periodically from the main loop, and `mns_receive_inic_message()` is called each time NSM runs. When `mns_receive_inic_message()` is called, it will:

- Check to see if the INIC **INT** pin is low
- If so, ask NetServices for an MBM buffer
- Get a pointer to the data buffer provided by MBM
- Read the message from INIC using the I^2C driver, putting the data into the buffer
- Call NetServices with the MBM buffer (`PmsRx()`)

Of course if the INIC **INT** pin is not low, `mns_receive_inic_message()` just returns without doing anything. The function called to actually read the message from INIC, `board_inic_read()`, is in *board.c* and it calls the `drv_i2c_read_buffer()` *function* in the *drv_i2c.c* file. The length parameter passed to `drv_i2c_read_buffer()` is 0, which is a special case which indicates that a Port Message should be read. Port message refers to the protocol used between NetServices and INIC. The first two bytes of a port message indicate the length of the message. With a length parameter of 0, the `drv_i2c_read_buffer()` function reads the first two bytes, calculates the length, then reads the rest of the bytes. The function returns the number of bytes in the message.

3.3.3 Adding Layer II

NetServices Layer II sits on top of Layer I and provides the main interface to the application for handling FBlocks. Layer II does not need to be initialized or serviced like Layer I. Layer I actually does that for us, which is why the *adjust1.h* file needed to know if Layer II is also included. Since there is no initialization structure for Layer II that can include the callback function pointers, callbacks are handled differently. Layer II calls specifically named functions that the application must implement. The NetServices example application distributed with NetServices includes a file, *ns_cb_l2.c* (NetServices Callback Layer II) which has stubs for all of the Layer II callbacks. The user must fill out the stub code for any callbacks that are needed. Like Layer I, each service of Layer II is in its own C source file, the major services are listed below.

- Command Interpreter – *cmd.c* – Interprets the incoming MOST network messages and calls an application function to handle the command.
- Address Handler – *ah.c* – Provides a network address lookup service so the application does not need to know the physical address of any devices on the network.
- Notification Service –*ntfs.c* – Provides the ability for an application to ‘announce’ changes to an FBlock’s property values. Prevents devices from having to poll to keep shadows up to date.
- NetBlock – *nbehc.c* – Each node must include NetBlock. Since its functions are defined by the MOST Specification 3.0 [2], NetServices can implement the entire FBlock using a few callbacks for the application to supply needed information.
- Enhanced Testability – *et.c* – An FBlock that provides an interface for external test tools to verify MOST network compliance. NetServices implements the basic FBlock and uses callbacks to the application for information or to perform some action.

Adding NetServices Layer II to an application involves the following steps, each of which will be explained in more detail below.

- Add all of the C source files in the Layer II folder to the project
- Edit the Layer II adjust file *adjust2.h* selecting the options needed by the application
- Edit the main FBlock table that lists each FBlock implemented by the device (*t_fblock.tab*)
- Add the tables that define each FBlock and its functions.
- Edit the main notification table for the device (*t_notify.tab*)
- Add the notification tables for each FBlock

3.3.3.1 LAYER II SOURCE FILES

There are 9 'C' source files in Layer II, and they should all be added to the project which is done the same way it was for Layer I. In the *MOST ToGo* project, these files are in /Source_Files/src/common/ns/L2.

3.3.3.2 LAYER II ADJUST FILE

The Layer II adjust file is used to select which services of Layer II the application needs to include. In addition to enabling these services, the Layer II adjust file has information about the FBlock tables. The Layer II adjust file, *adjust2.h,* is located in a sub-folder inside the L2 folder, *L2/UserAdjustable*.

Chapter 4 of the MOST NetServices Layer II User's Guide [14] describes each setting in the *adjust2.h* file. The *adjust2.h* file is organized in sections that roughly correspond to the major services of Layer II outlined above. Likewise, Chapter 4 is organized by these same sections, outlining the options for each service.

The complete settings of the *adjust2.h* file for the *MOST ToGo* basic application can be seen in the *MOST ToGo* Code Documentation help file. A few of the major settings are given below.

- The command interpreter is enabled
- NetServices' implementation of NetBlock is enabled (users can implement their own version)
- The address handler is enabled (for the Master application only)
- `NETWORKMASTER_LOCAL` is defined for the master application, while the slaves define `NETWORKMASTER_SHADOW`
- The notification service is enabled
- NetServices' implementation of FBlock `EnhancedTestability` is enabled (the user could implement their own version)

3.3.3.3 MAIN FBlock TABLE

The FBlocks on a device are listed in a table that the command interpreter needs. This file is called *t_fblock.tab* and is found in the L2/UserAdjustable folder. A shell of the table is delivered with NetServices with the FBlocks that NetServices provides already there. The user must edit *t_fblock.tab* and add the FBlocks and shadows that the application will create.

The FBlock table from the edited *t_fblock.tab* file for the sink application (`AudioAmp`) is shown in Example 3-5 Main FBlock Table For Sink Application:

© 2014 Microchip Technology Inc.

EXAMPLE 3-5: MAIN FBlock TABLE FOR SINK APPLICATION

```
_CONST struct FBlock_L_Type FBlocks[CMD_SIZE_FBLOCK_TABLE] =
  {
      { FBLOCK_NETBLOCK,  DONT_CARE,            &Func_NetBlock[0]       }   /* NetBlock */
     ,{ FBLOCK_AMP,       PTR_INSTID(IDX_AMP),  &Func_Amp[0]            }   /* Amplifier */
     #ifdef ET_MIN
     ,{ FBLOCK_ET_ID,     DONT_CARE,            &Func_ET[0]             }   /* FBlock ET  */
     #endif
     #ifdef NETWORKMASTER_LOCAL
     ,{ FBLOCK_NWM,       PTR_INSTID(IDX_NWM),  &Func_NetworkMaster[0]  }   /* NetworkMaster */
     #endif

    /* following FBlocks can receive report messages only (FBlockShadows)*/

     #ifdef NETWORKMASTER_SHADOW
     ,{ FBLOCK_NWM_SHDW,  DONT_CARE,           &Func_NetworkMaster_S[0] } /* NetworkMaster Shadow */
     #endif
  };
```

As can be seen, the table has 3 columns, the first is the FBlock ID, the middle column is the InstID (actually a pointer to the InstID), and the third is a pointer to another table that contains a list of all of the functions contained in the FBlock. That table is in another *.tab* file, usually named for the FBlock it defines. We will see an example of one of these in the next section.

FBlock `AudioAmp` defines 3 FBlocks, `FBLOCK_NETBLOCK`, `FBLOCK_AMP`, and `FBLOCK_ET` which is included because `ET_MIN` was defined in *adjust2.h*. `FBLOCK_NWM` is not defined because this is a slave node and `NETWORKMASTER_LOCAL` was not defined in *adjust2.h*. Finally, `FBLOCK_NWM_SHDW` will be defined because the AudioAmp device is not a master node so `NETWORKMASTER_SHADOW` is defined in *adjust2.h*. So, the AudioAmp device ends up with 3 FBlocks and 1 FBlock shadow. Of these FBlocks, NetServices implements `FBLOCK_NETBLOCK`, `FBLOCK_ET`, and `FBLOCK_NWM_SHDW`, leaving the user only `FBLOCK_AMP` to implement – thank you NetServices!

Compare that to the FBlock table from the master device's *t_fblock.tab* file. See Example 3-6 Main FBlock Table For Master Application:

EXAMPLE 3-6: MAIN FBlock TABLE FOR MASTER APPLICATION

```
CONST struct FBlock_L_Type FBlocks[CMD_SIZE_FBLOCK_TABLE] =
    {
      { FBLOCK_NETBLOCK,     DONT_CARE,                &Func_NetBlock[0] }   /* NetBlock */
     ,{ FBLOCK_AMP,          PTR_INSTID(IDX_AMP),      &Func_Amp[0]      }   /* Amplifier FBlock */
     ,{ FBLOCK_AUXIN,        PTR_INSTID(IDX_AUXIN),    &Func_Auxin[0]    }   /* Aux In FBlock */
     ,{ FBLOCK_HMI,          PTR_INSTID(IDX_HMI),      &Func_HMI[0]      }   /* HMI FBlock */
     ,{ FBLOCK_CONMASTER_ID, PTR_INSTID(IDX_CONMASTER), &Func_MSM_CM[0]  }   /* MSMM Connection Master */

     #ifdef NETWORKMASTER_LOCAL
     ,{ FBLOCK_NWM,          PTR_INSTID(IDX_NWM),&Func_MSM_NWM[0]        }  /* NetworkMaster */
     #endif

     #ifdef ET_MIN
     ,{ FBLOCK_ET_ID,        PTR_INSTID(IDX_ET), &Func_ET[0]             }  /* FBlock ET  */
     #endif

    /* following FBlocks can receive report messages only (FBlockShadows):                */

     #ifdef NETWORKMASTER_SHADOW
     ,{ FBLOCK_NWM_SHDW,     DONT_CARE,          &Func_NetworkMaster_S[0]}   /* NetworkMasterShadow, */
     #endif

    ,{ FBLOCK_AMP,           DONT_CARE,          &Func_Amp_Shdw[0]       }   /* Amplifier FBlock Shadow*/
    ,{ FBLOCK_AUXIN,         DONT_CARE,          &Func_Auxin_Shdw[0]     }  /* Aux In FBlock Shadow*/

    };
```

We see that the master application has 7 FBlocks defined (NETWORKMASTER_LOCAL is defined in *adjust2.h* so FBLOCK_NWM is defined), and 2 shadows. This also shows how a device can have an FBlock and a shadow of the same function (FBLOCK_AMP and FBLOCK_AUXIN), and how they are defined

© 2014 Microchip Technology Inc.

3.3.3.4 INDIVIDUAL FBlock TABLES

For each FBlock defined in the *t_fblock.tab* table, there must be a corresponding table containing a list of the functions of the FBlock and its methods and properties. Example 3-7 Function Table for FBlock AudioAmp shows FBlock `AudioAmp`. This table is defined in *t_fblock_amp.tab* which is located with the main source files of the application.

EXAMPLE 3-7: FUNCTION TABLE FOR FBlock `AudioAmp`

```
const struct Func_L_Type Func_Amp[] =
{                                                                                    //
    {FUNC_FKTIDS,                  &Op_Amp_FuncIDs[0]             },                 //
    {FUNC_NOTIFICATION,            &Op_Amp_Notification[0]        },                 //
    {FUNC_NOTIFICATIONCHECK,       &Op_Amp_NotificationCheck[0]   },                 //
    {FUNC_FBLOCKINFO,              &Op_Amp_FBlockInfo[0]          },                 //
    {FUNC_SINKINFO,                &Op_Amp_SinkInfo[0]            },                 //
    {FUNC_CONNECT,                 &Op_Amp_Connect[0]             },                 //
    {FUNC_DISCONNECT,              &Op_Amp_DisConnect[0]          },                 //
    {FUNC_MUTE,                    &Op_Amp_Mute[0]                },                 //
    {FUNC_SINKNAME,                &Op_Amp_SinkName[0]            },                 //
    {FUNC_SYNCDATAINFO,            &Op_Amp_SyncDataInfo[0]        },                 //
    {FUNC_AMP_VOLUME,              &Op_Amp_Volume[0]              },                 //
    {FUNC_AMP_BASS,                &Op_Amp_Bass[0]                },                 //
    {FUNC_AMP_TREBLE,              &Op_Amp_Treble[0]              },                 //
    {FUNC_AMP_TEMPERATURE,         &Op_Amp_Temperature[0]         },                 //
                                                                                     //
    {FUNC_TERMINATION,             0                              }                  //
};                                                                                   //
```

The first thing to notice is that the name of this table is *Func_Amp* which is the name given in *t_fblock.tab* in the third column for FBlock `AudioAmp` (same in either `AudioAmp` or master application). Second, the functions listed here are what were defined in the function catalog when FBlock `AudioAmp` was designed. The values of the macros like `FUNC_AMP_VOLUME` are also defined in the function catalog. In fact, compare this table to the function catalog entry shown in Figure 2-3 and notice the listing of functions there for FBlock `AudioAmp`.

The third thing to notice in this 2 column table is that the 2nd item points to another table. These tables are called the OpType tables and they are included in the same tab file as the function table above. Those tables define all of the OpTypes defined for each function, so there is an OpType table for each function. Example 3-8 OpType Table For FBlock AudioAmp shows the OpType table for the Volume function.

EXAMPLE 3-8: OpType TABLE FOR FBlock `AudioAmp`

```
//-------------------------------------------------------------------------------------------------------//
//   Table of available Operations of     FBlock:   Amplifier                                            //
//                                        Function: Volume                                               //
//-------------------------------------------------------------------------------------------------------//
//   OP_TYPE   |  Flags   |   Ptr for write access   |   Ptr for read access   |  Length Check          //
//-------------------------------------------------------------------------------------------------------//
//                                                                                                       //
const struct Op_L_Type Op_Amp_Volume[] =                                                               //
{                                                                                                      //
  {OP_SET,          0xD0,    NS_F_V amp_volume_set,     0,                              LC_EQ(1)  },     //
  {OP_GET,          0x04,    0,                         NS_F_V &amp.prop_volume,        LC_NO     },     //
  {OP_SETGET,       0xD4,    NS_F_V amp_volume_set,     NS_F_V &amp.prop_volume,        LC_EQ(1)  },     //
  {OP_INC,          0xD4,    NS_F_V amp_volume_inc,     NS_F_V &amp.prop_volume,        LC_EQ(1)  },     //
  {OP_DEC,          0xD4,    NS_F_V amp_volume_dec,     NS_F_V &amp.prop_volume,        LC_EQ(1)  },     //
  {OP_TERMINATION,  0,       0,                         0,                              LC_NO      }     //
};                                                                                                     //
//-------------------------------------------------------------------------------------------------------//
```

The OpType tables have five columns which are:

- OpType ID (defined in the Fcat schema),
- Flag field, a byte whose upper nibble determines the type of write function, and whose lower nibble determines the type of read function. These flags are defined in the Layer II User Manual in Table 7-3 and Table 7-4 respectively.
- a pointer to a set type function (write), this function belongs to the FBlock implementation and is part of the application itself
- a pointer to a get type function (read), or a pointer to a property (depending on Flag field setting), the property or function is part of the application itself
- Length check macro – NetServices command interpreter can automatically check the length of the parameters passed in if this feature is enabled.

It is interesting to compare this table to Figure 2-3 where the OpType table is defined. It is easy to see how the OpType table above can be created directly from the function catalog definition.

The pointers in the OpType table point to functions or properties that are in the code that implements the FBlock itself (*fblock_amp.c* in this case). The FBlock code must implement every function that gets defined in the OpType table. So the OpType table defines the functions that get called and, via the flag field, the prototypes of those functions. The contents of the OpType table were defined back at the function catalog creation stage. That is why it was said that the function catalog determines the API of the function block.

For the complete information on the definition of these tables used by the command interpreter, see Chapter 7 of the MOST NetServices Layer II User's Guide [14].

© 2014 Microchip Technology Inc.

3.3.3.5 MAIN NOTIFICATION TABLE

The notification service that NetServices provides allows one device to subscribe to property change events of another device, a ‘publish and subscribe’ paradigm. This service prevents devices from having to poll other devices in order to keep up to date with property values. The notification service is described in Chapter 12 of the MOST NetServices Layer II User’s Guide [14].

When implementing an FBlock, the properties that can publish their values are defined in a couple of tables. The first table is for the overall device and it just lists the FBlocks that support notification. Example 3-9 Notification Table for MOST ToGo Master Application shows the table for the *MOST ToGo* master application. It is contained in the *t_notify.tab* file and is found in the L2/UserAdjustable folder.

EXAMPLE 3-9: NOTIFICATION TABLE FOR MOST ToGo MASTER APPLICATION

```
/*-------------------------------------------------------------------------------------------------------*/
/*     Table of all FBlocks,                                                                             */
/*     which have to be serviced by the Notification Service                                             */
/*                                                                                                       */
/*-------------------------------------------------------------------------------------------------------*/
/*  FBlock  |   Maximum number of   |   Ptr at                                                            */
/*  Index   |   devices to notify   |   property table                                                    */
/*-------------------------------------------------------------------------------------------------------*/
/*                                                                                                       */
TNtfFBlockL NtfFBlockTab[NTF_MAX_FBLOCKS] =
    {
      { 0,   NTF_SIZE_DEVICE_TAB,   &NtfPropTabAmp[0]              }, /* FBlock Amp         index = 0)  */
      { 1,   NTF_SIZE_DEVICE_TAB,   &NtfPropTabAuxin[0]            }, /* FBlock Auxin       index = 1)  */
      { 2,   NTF_SIZE_DEVICE_TAB,   &NtfPropTabConnectionMaster[0] }, /* FBlock ConMstr     index = 2   */

    };
/*-------------------------------------------------------------------------------------------------------*/
```

The master application has 3 FBlocks that can support notification. The pointers in the third column are to another table that defines which properties of the FBlock support notification. Those tables are in the FBlock tables along with Function and OpType tables described above.

3.3.3.6 INDIVIDUAL NOTIFICATION TABLES

Each FBlock that supports notification must have a table which specifies which properties support notification and how to access those properties. The notification table for FBlock `AudioAmp` is shown in Example 3-10 FBlock AudioAmp Notification Table. As mentioned above, it is located in *t_fblock_amp.tab*.

EXAMPLE 3-10: FBlock `AudioAmp` NOTIFICATION TABLE

```
//-------------------------------------------------------------------------------------------------------//
//   Table of all Properties of FBlock                                                                   //
//   which have to be serviced by the Notification Service                                               //
//                                                                                                       //
//-------------------------------------------------------------------------------------------------------//
//     Flags / FuncID      |      Base Addr of         | Pointer at |   Pointer at                       //
//                         |      Notification         | Property   |   Property                         //
//                         |      Matrix               |            |   Reference                        //
//-------------------------------------------------------------------------------------------------------//
//                                                                                                       //
TNtfPropL NtfPropTabAmp[] =                                                                             //
 {                                                                                                      //
    { NTF_IND  | FUNC_SINKINFO,       &ntf_matrix_amp[0][0],  NS_F_V &amp_sink_info_get_notif  },       //
    { NTF_IND  | FUNC_SINKNAME,       &ntf_matrix_amp[1][0],  NS_F_V &amp_sink_name_notif      },       //
    { NTF_IND  | FUNC_AMP_MUTE,       &ntf_matrix_amp[2][0],  NS_F_V &amp_mute_get_notif       },       //
    { NTF_BYTE | FUNC_AMP_VOLUME,      &ntf_matrix_amp[3][0], NS_F_V &amp.prop_volume          },       //
    { NTF_BYTE | FUNC_AMP_TEMPERATURE, &ntf_matrix_amp[4][0], NS_F_V &amp.prop_temperature     },       //
    { NTF_TERMINATION,                 0,                     0                              }        //
 };                                                                                                     //
//-------------------------------------------------------------------------------------------------------//
```

The first column consists of a flag field and the Function ID (FktID) of the property. The flag field indicates whether the notification service should call an individual function (IND), or if a pointer to a property is provided. The second column has a pointer to the notification matrix which is an array that the notification service maintains. The third column points to a function that will get called or points to a property.

When the application changes a property that supports notification, it must call the notification service to announce the property change. The notification service will then 'publish' this information by sending a status message to any devices that may be subscribed to this property. Example 3-11 Notification Service Trigger shows the line of code from the *volume_set()* function of FBlock `AudioAmp` shows how the notification service is triggered:

EXAMPLE 3-11: NOTIFICATION SERVICE TRIGGER

```
       NtfPropertyChangedFkt(FBLOCK_AMP, 0, FUNC_AMP_VOLUME);
```

Note that the master application has 3 FBlocks that support notification, while there are 7 FBlocks defined for the application. Also, FBlock `AudioAmp` supports 5 properties for notification, while the function table defined 9 properties (methods do not support notification). This brings up two important points: every FBlock is not required to support notification; every property of an FBlock is not required to support notification. The designer of the FBlock determines which properties should support notification (yes, defined way back in the function catalog creation).

With the Layer II source files included in the project, and the *adjust2.h* file properly modified, and the command interpreter and notification tables all filled in, the incorporation of Layer II is complete. Now we can move on to implementing the FBlock itself, defining the routines that the command interpreter will call.

© 2014 Microchip Technology Inc.

3.4 CREATING FBlocks

The next step in creating the application is to implement the functions that make up the FBlock. These functions were actually defined during the function catalog creation stage, but the FBlock tables used by the command interpreter define the actual name and form of the function based on the settings of the flag field. FBlocks can contain two types of functions, properties and methods. The basic mechanisms work the same for either, but there are some differences. First we will look at implementing a property.

3.4.1 Properties in FBlocks

The first thing to do is create a stub function for each function defined, for both properties and methods. Example 3-12 Creating Stub Function for Amp.Volume.Set shows what the *Amp.Volume.Set* function would look like:

EXAMPLE 3-12: CREATING STUB FUNCTION FOR *Amp.Volume.Set*

```
uint8_t amp_volume_set(TMsgTx *tx_ptr, TMsgRx *rx_ptr)
{
    return (OP_NO_REPORT); // no response to .Set
}
```

The flags selected in the flag field in the command interpreter table entry for this function specifies that the command interpreter will send a pointer to the received message (the command) and a pointer to a transmit message (for a reply). The received message is required so that we can get the parameter value which is the volume setting in this example. Even though the *Set* function doesn't normally send a reply back to the sender, a reply message may be needed in case of an error.

The functions that the command interpreter calls (like `amp_volume_set()`) all return a parameter back to the command interpreter. The common values for properties are:

- `OP_NO_REPORT` – no reply message is needed
- `OP_STATUS` – a reply is sent back to the sender indicating success
- `OP_ERROR` – a reply is sent back to the sender indicating an error occurred.

If an *fblock_amp.c* file is created with stub functions for all the functions defined in the command interpreter table, then the application should compile OK; the compiler will complain of any forgotten functions. The application should even run, but it won't do any useful work since all the functions are empty.

Once the application is compiling OK, it is time to go back and add the code to actually do the work. For most functions this usually involves the following steps:

- Extract the parameter from the received message, if any
- Do any required range checking on the parameter if applicable
- Do the work required of this function (e.g. actually change the volume of the CODEC)
- Build the reply if needed

Example 3-13 Final Version of amp_set_volume Function shows the final version of the `amp_volulme_set()` function.

EXAMPLE 3-13: FINAL VERSION OF `amp_set_volume` FUNCTION

```
uint8_t amp_volume_set(TMsgTx *tx_ptr, TMsgRx *rx_ptr)
{
    // Parameter checking - 1 parameter - volume setting
    if (rx_ptr->Data[0] > VOLMAX)
        return (CmdErrorParamWrong(tx_ptr, 1, &rx_ptr->Data[0], 1));

    amp.prop_volume = rx_ptr->Data[0]; // Get the Volume from message
    volume_set();
    return (OP_NO_REPORT); // no response to .Set
}
```

If the volume value passed in fails the parameter check, the `CmdErrorParamWrong` helper function provided by the command interpreter will build an error message and return it to the sender. Otherwise, the volume setting is extracted and saved in a structure the application maintains. Then the `volume_set()` function is called which actually changes the volume. The `volume_set()` function also triggers the notification service as we saw earlier.

Next we will look at a function that does return a value to see how that works. Example 3-14 amp_volume_get() Function shows the `amp_volume_get()` function, which is sort of the compliment to `amp_volume_set()`.

EXAMPLE 3-14: `amp_volume_get()` FUNCTION

```
uint8_t amp_volume_get(TMsgTx *tx_ptr, TMsgRx *rx_ptr)
{
    tx_ptr->Data[0] = amp.prop_volume; // Read the Volume
    tx_ptr->Length = 1;
    return (OP_STATUS);
}
```

Note there are no parameters to the *Get* function, therefore parameter extraction and range checking are not required. The current volume value, which is a single byte, is simply put in the reply message parameter section. The return of `OP_STATUS` causes the message to be returned to whoever asked.

So in summary, if a device sent the message *Amp.Volume.Set.12* to our FBlock, the `amp_volume_set()` function would set the volume to 12 (assuming 12 is less than `VOLMAX`). Now, if the device sent the message *Amp.Volume.Get*, our `amp_volume_get()` function would send back the message *Amp.Volume.Status.12*.

© 2014 Microchip Technology Inc.

3.4.2 Methods in FBlocks

The main difference between properties and methods in a MOST system, is that a property has a current value and it can be queried and returned immediately. A method is used to control a process and the result may not be available immediately. Therefore methods need to be able to start an action, and reply at a later time. Similarly, controllers need to know that they may have to wait for the result. Working with methods often involves the use of state machines to handle this asynchronous nature.

Implementing a method in an FBlock is similar to implementing a property. The definition looks the same in the command interpreter table and the function prototypes are the same. Methods do use different OpTypes and responses, for example `OP_START` instead of `OP_SET`, and `OP_RESULT` instead of `OP_STATUS`.

Example 3-15 amp_connect_start_result() function shows the *Connect* method from FBlock `AudioAmp`. Its purpose is to connect a streaming source from the network and route it to an INIC streaming port pin that presumably is connected to a D/A converter. Connecting to a network channel is a multi-step process:

- Open the input port (not necessary if the port is MOST network, it is always open)
- Open the output port (streaming port in this case)
- Create a socket on the input port
- Create a socket on the output port
- Connect the sockets

The application will have to call the Socket Connection Manager (SCM) of Layer I to do these tasks, and each one of them is itself a method on INIC. Therefore the application will have to wait for the results from INIC before proceeding to the next step.

In the property section there was a four step process outlined to handle property functions which is repeated here:

- Extract the parameter from the received message, if any
- Do any required range checking on the parameter if applicable
- Do the work required of this function
- Build the reply if needed

For methods the steps are slightly modified:

- Extract the parameter from the received message, if any
- Do any required range checking on the parameter if applicable
- Prepare to reply in the future when process is done
- Kick off the process to do the work
- Wait for the process to finish
- Reply to the sender

© 2014 Microchip Technology Inc.

EXAMPLE 3-15: `amp_connect_start_result()` **FUNCTION**

```
uint8_t amp_connect_start_result(TMsgTx *tx_ptr, TMsgRx *rx_ptr)
{
    uint16_t new_connection;
    uint16_t block_width;

    /* parameters:  SenderHandle (word  0:1)
                    SinkNr, (byte) 2
                    BlockWidth (word) 3:4
                    ConnectionLabel (word) 5:6
    */
    if (NULL != amp_retained_tx_ptr)
    {
        /* if not null, currently working on some other request */
        return (CmdErrorMsg(tx_ptr, ERR_BUSY));
    }
    if (rx_ptr->Data[2] != AMP_SINK_NR)
    {
        return(CmdErrorParamWrong(tx_ptr, 2, &rx_ptr->Data[2], 1));
    }
    /* extract paramters */
    DECODE_WORD(&sender_handle, &rx_ptr->Data[0]);
    DECODE_WORD(&block_width, &rx_ptr->Data[3]);
    DECODE_WORD(&new_connection, &rx_ptr->Data[5]);
    TRACE_T("Connection Label for Stereo Amp is: 0x%X\n\r", new_connection);

    if (AUDIO_STEREO*RES_16BIT != block_width)
    {
        return (CmdErrorParamWrong(tx_ptr, 3, &rx_ptr->Data[3], 2));
    }

    // Command OK
    /* prepare as much of answer as possible, assuming things go OK */
    tx_ptr->Length = 1;
    tx_ptr->Operation = OP_RESULTACK;
    tx_ptr->Data[0] = AMP_SINK_NR;

    /* keep pointer to return message - returned later */
    amp_retained_tx_ptr = tx_ptr;

    /* now request nsm to make the connection */
    nsm_request_connection(AMP_CON, CRS_OPEN, FBLOCK_AMP, new_connection);
    /* nsm will fire callback amp_on_event when done where reply will be completed */

    return (CMD_TX_RETAIN);

}
```


© 2014 Microchip Technology Inc.

The first check is to see if a pointer, `amp_retained_tx_ptr` is in use. The purpose of this pointer will be explained a little later. After that, the normal parameter checking is done, and errors could be generated. If all parameters are OK, then as much of the reply message as possible is filled out, here assuming that the method will succeed. However, the reply cannot be sent yet, as the work has not been done. This is where the retained pointer comes in, a copy of the pointer to the reply message is kept in `amp_retained_tx_ptr` for later use.

Now the actual action is started. As mentioned, many methods require a state machine, and especially this one as it involves several steps to be carried out by the Socket Connection Manager (SCM) on INIC as outlined above. Creating (or destroying) connections like this is one of the main functions of the NSM (NetServices State Machine) module that has been mentioned before.

The call to `nsm_request_connection()` tells the NSM that it should create a connection (`CRS_OPEN`) defined in a table by the `AMP_CON` label, using the connection label (channel) in `new_connection`. When done, NSM should notify `FBLOCK_AMP`. The NSM is described in more detail in Section 3.5 “NSM - NetServices State Machine”.

At this point, the process is started by NSM, and the method should return to NetServices. Here the special return value `CMD_TX_RETAIN` is used which tells NetServices that we are not done with the buffer containing the transmit (reply) message that was passed in, and that NetServices should not to send the reply yet.

The NSM module will now do its work in the background, opening ports, creating and connecting the required sockets. When done it will fire an event that will eventually get handled by `amp_on_event()` in the *fblock_amp.c* module. The event handler function will then finish filling out the reply using the retained pointer and send the reply back. It will also trigger notification of the SINKINFO property so that the Connection Master is notified of the new status of the connection. The `amp_retained_tx_ptr` value is set to NULL indicating the operation is done, and more importantly that the socket connection manager on INIC is free again.

© 2014 Microchip Technology Inc.

3.5 NSM - NetServices STATE MACHINE

The NetServices state machine or NSM *(nsm.c)* is a utility routine that simplifies the interaction between NetServices and the application. The NSM module provides four main utilities:

- Initialization of NetServices
- Periodic servicing of NetServices
- Socket Connection Manager State Machine
- NetServices Layer I event handling

3.5.1 Initializing NetServices

As described in Section 3.3.1.4 “Initializing NetServices” there is a large structure that needs to be filled out in order to initialize NetServices. That structure is in the NSM module. A call to `nsm_init()` will fill out the structure and call the `InitNetServices()` function. The `nsm_init()` routine is called from t*ask_most.c*, from its `init()` routine. The structure has several pointers to callback functions, and most of those functions are also in the NSM module.

3.5.2 Servicing NetServices

As described in Section 3.3.1.5 “Servicing NetServices” NetServices needs to be called periodically so it can do its work. Both the event driven calls to `MostService()` and the time based calls to `MostCheckTimers()` are handled inside of the NSM module with a single call to the NSM service function `nsm_service()`. In the *MOST ToGo* implementation, `nsm_service()` is called from `task_most()` which is called from the main loop.

3.5.3 Socket Connection Manager State Machine

Building and destroying connections from INIC to resources like streaming ports, the network port, or MediaLB port is one of the more complicated tasks that the application must do. As pointed out in Section 3.4.2 “Methods in FBlocks” building connections is a multi-step process that generally requires a state machine. NSM provides that state machine for building (or destroying) these connections.

The NSM module uses a table that specifies which ports the connections use and the parameters for the ports, and another table for the connections with parameters for the sockets the connection will use. These tables are in *nsm.usr.* Below is an excerpt from the *nsm.usr* table of the description of the streaming port which is used for both the `AudioAmp` and `AuxIn` applications.

© 2014 Microchip Technology Inc.

EXAMPLE 3-16: NSM PORT TABLE

```
port_t    Port[MAX_PORT] =
{
    {
    // ------------------------------------------------- //
    // port 0                                            //
    // ------------------------------------------------- //
       {
            .port_id                          = SCM_PORT_ID_STREAM,
            .config.streaming.clock_drive_mode = SCM_PORT_CFG_STREAM_SCKFSY_INIC,
            .config.streaming.port_mode       = SCM_PORT_CFG_STREAM_FULL,
            .config.streaming.data_format     = SCM_STREAM_FRMT_DEL_64FS_16BIT,
        }
     }
};
```

For all of the *MOST ToGo* basic applications, only the streaming port and network port are used. The network port (MOST Port) is always open when the ring is locked and needs no configuration. Therefore the only port that needs to be configured is the streaming port, and that is why there is only one entry in the port table. The `SCM_...` constants are all available when *mostns.h* is included in a file.

The parameters that are listed for the streaming port are defined in Chapter 13 of the MOST NetServices Layer I User's Guide [13], and in even more detail in the INIC API User Manual for the respective INIC.

The second table in *nsm.usr* is the connection table which describes all of the connections the application might use. Below is an excerpt from *nsm.usr* for the amplifier connection.

EXAMPLE 3-17: NSM CONNECTION TABLE

```
connection_t    Con[MAX_CON] =
{
    // ------------------------------------------------- //
    // connection 0 - Audio Amp                          //
    // receive Audio from MOST and route it to SX0       //
    // ------------------------------------------------- //
    {
        .port_rx = 0,
        .port_tx = 0,
        {
            .port_id                    = SCM_PORT_ID_MOST,
            .direction                  = SCM_IN,
            .datatype                   = SCM_TYPE_SYNC,
            .blockwidth                 = 4,
            .most.flags                 = SCM_MOST_FLAGS_DEFAULT,
        },
        {
            .port_id                    = SCM_PORT_ID_STREAM,
            .direction                  = SCM_OUT,
            .datatype                   = SCM_TYPE_SYNC,
            .blockwidth                 = 4,
            .streaming.interface_id     = SCM_STREAM_INTERFACE_SX0,
            .streaming.offset           = 0,
        }
    },
```

The first elements define which ports are used for the In and Out data streams. It looks like both are set to stream port since they are both set to 0, but NSM will see that the `port_id` is *SCM_PORT_ID_MOST* and know that it does not need to be opened. NSM keeps up with which ports are opened, and if another connection wants to use the same port, NSM will know whether or not the port is already opened. Note that the streaming ports can support up to 4 connections. Similarly when NSM is destroying sockets on ports, it will know if no other connections are using a port, then that port can also be closed.

Next there are two subsections which contain the parameters for the In and Out sockets of the connection. Each subsection has all the parameters that are needed for type and direction of the socket as defined in the Socket Connection Manager section of the respective INIC API User Manual. The *nsm.usr* file will have an entry like this for each connection in the system. In the *MOST ToGo* applications, the Source (`AuxIn`) and Sink (`AudioAmp`) applications have only one connection in the table, while the HMI application has two, one for the AuxIn connection and one for its `AudioAmp` connection.

With all of the parameters for the ports and sockets in the tables defined, the user application can create connections with a single call to `nsm_request_connection()`. See Example 3-18 Excerpt from amp_connect_start_result() function for the example from *fblock_amp.c* that was seen earlier in Example 3-15 amp_connect_start_result() function.

EXAMPLE 3-18: EXCERPT FROM `amp_connect_start_result()` FUNCTION

```
    /* now request nsm to make the connection */
    nsm_request_connection(AMP_CON, CRS_OPEN, FBLOCK_AMP, new_connection);
    /* nsm will fire callback amp_on_event when done where reply will be completed */
```


© 2014 Microchip Technology Inc.

As can be seen, there are 4 parameters to the function. The first is which connection is involved, it should be the index into the connection table in *nsm.usr*. Here `AMP_CON` is defined to be 0, which is the index of the amplifier connection in the table. The second parameter should be either `CRS_OPEN` to create a connection or `CRS_CLOSE` to destroy an existing connection. The third parameter is called the user parameter, it is returned to the user when NSM is done with the connection and fires an event. Here the FBlock ID is used to distinguish which FBlock had requested the action. The fourth parameter is the *ConnectionLabel* of the connection. On a sink connection, such as the amplifier, this must be passed in, and in this example the variable `new_connection` had earlier been set to the *ConnectionLabel* that was passed in via the command interpreter. For a source connection like the AuxIn device, this parameter should be set to NULL, the *ConnectionLabel* that an Allocate receives will be extracted when the process is done. Once any connection is completed, the application can call `nsm_get_connection_label()` with the index to the connection, and NSM will return the *ConnectionLabel*.

Once NSM is started with the `nsm_request_connetion()` it will work in the background to open/close ports, create/destroy sockets, and create/destroy connections until the process is complete. It will then fire an event to the event handler (explained in next section) with an event type of `ET_USER` and a parameter of `user_parameter` which is passed in to the function. The event handler should route this result back to whoever started the action (FBlock `AudioAmp` in this case).

3.5.4 Layer I Event Handling

In the NetServices initialization section there were a lot of callbacks to be implemented by the application, and pointers to those callbacks are filled out in the initialization table. These callbacks are all defined in the NSM module. While they are defined here, NSM does not know how to handle most of them, they need to be passed on to the application. NSM defines an event handler and turns most of the callbacks (a few are handled locally) into an event type and a parameter, and then calls the event handler which the user application provides. In the *MOST ToGo* applications, that event handler is in *task_most* and is `app_on_nsm_event()`. In *task_most*, a CASE statement handles the events by type, and usually also passes them on to other application tasks. This is how most of the Layer I supervisor events get routed to the parts of the application that need to handle them.

© 2014 Microchip Technology Inc.

3.6 INTEGRATING MSMM

The MOST System Management Module or MSMM is a set of sub modules that sit on top of NetServices and provide the handling of system management tasks. Specifically MSMM provides the Network Master, Connection Master and Power Master functions for a device Note that these functions are typically associated only with the network master device, The Network Master and Power Master blocks are required to be in the master device by the MOST Specification 3.0 [2]. It is highly recommended to place the Connection Master on the same device as the Network Master so they can share the Central Registry. MSMM also includes an enhanced connection master called the Extended Connection Manager, but the basic *MOST ToGo* System does not need those enhanced capabilities.

MSMM is delivered as ANSI C source code just like NetServices. There are 5 folders:

- ConnectionMaster - files for both the standard Connection Master and Extended Connection Master, also the *.tab* file to implement the FBlock
- External_Interface_Files - Callback functions that the user must implement
- MSM_Service - provides the supervisor for MSMM
- NetworkMaster - provides the code for implementing FBlock `NetworkMaster` and its *tab* file for the command interpreter
- PowerMaster - code to implement power master functionality, note that the power master is not a function block

The MOST System Management Module (MSMM) [3] contains step by step directions to add MSMM to an application that is based on NetServices. Like Layer I and Layer II of NetServices, MSMM has an adjust file (*adj_msm.h*) that is used for configuring the different services of MSMM. The file contains a section for each MSMM service. The manual will lead the user step by step through each section adding each layer in turn. There are chapters for:

- MSMM Service Module - the kernel of MSMM needed by every service
- Power Master Module - configuring network startup and shutdown
- Network Master Module - adds FBlock `NetworkMaster`, the Network Master manages the Central Registry and ensures the network is configured properly, notifies all nodes about network configuration via the `NetworkMaster.ConfigStatus` message
- Connection Master Module - adds FBlock `ConnectionMaster` which handles synchronous (and isochronous for MOST150) connections between devices.

For each section the manual outlines the steps needed to configure each module. These steps usually include:

- The parameters of *adj_msm.h* that need to be defined for the module
- The API the module provides to the application
- Callbacks the module defines that the application needs to implement. Instructions for the implementation required are given.
- Settings of Layer I *adjust1.h* that are needed by MSMM
- Settings of Layer II *adjust2.h* that are needed by MSMM
- For the modules that add FBlocks to the application, (Network Master and Connection Master) instructions are given on what to add to the command interpreter tables (*.tab* files).

© 2014 Microchip Technology Inc.

MSMM is included in the *MOST ToGo* master project in the HMI module, and the code is located in the project virtual folders under *Source Files/src/common/msmm/*. The *adj_msm.h* file is in the *MSM_Service* folder, and the callback routines that MSMM requires are in the *msm_cb.c* file in the *External_Interface_File* folder. Note that MSMM needs to be notified about several of the Layer I events. Each of these is specified by the MSMM manual. These will be found in the event handler in *task_most* which is triggered by the NSM module where the Layer I callbacks are handled as discussed in the previous section.

Note that MSMM PowerMaster module has settings for automatic network startup at initialization, automatic retries on network startup failure, and triggering of RingBreakDiagnostics on startup failure. These features are not used in the *MOST ToGo* application because this version of MSMM does not support the full ECL (Electrical Control Line) wakeup and diagnostics defined in the ECL specification. Instead, all of the ECL logic and network wakeup and startup logic is implemented in the PMGT module of the *MOST ToGo* applications. The PMGT module then makes direct calls to MSMM to start or shutdown the network.

However, in the near future a new version of MSMM will be released that does support the full ECL and System Tests diagnostics specifications. At that time, the *MOST ToGo* applications will be restructured to take advantage of the new MSMM ECL modules, and greatly simplify the logic in the current PMGT module.

3.7 APPLICATION LOGIC

Once the basic embedded application code that controls the application hardware (such as an amplifier, or FM Tuner), is done, NetServices and MSMM are integrated, and the FBlocks defined in the function catalog are implemented, the only thing left is the top level application logic of the device. This logic is what gives the device its whole look and feel, and controls its overall operation. This domain should be the main area of expertise provided by the OEM and their key Tier Ones. As mentioned before, the *MOST ToGo* applications try to keep this part of the application as simple as possible so as not to confuse the issues with creating a MOST system-enabled device.

The basic *MOST ToGo* applications are implemented in a main loop architecture, a so called ‘while 1’ operating system. However, the system is organized as if a Real Time Operating System (RTOS) is used. There are 5 main tasks that are called repeatedly from the main loop. The entire *main.c* module simply has some initialization to get the CPU clocks going, initialize the GPIO system, then it falls into the main loop where the 5 tasks are repeatedly called.

The 5 tasks are each in their own module and are:

- Board Task - *task_board.c* - keeps the state of all GPIO inputs up to date in the background, maintaining a shadow of the current state. Other modules that want to check the state of a pin have the option of just reading the current shadow, or triggering a new GPIO read.
- UI Task - *task_ui.c* - handles the user interface including switches and LEDs on the board as well as the basic system display (if attached). The switches include the 8 DIP switches, the rotary switch, the individual pushbuttons labeled SWA and SWB. The on-board LEDs include the 3 application LEDs, one is used as a message LED that blinks each time a MOST network message arrives.
- MOST Task - *task_most.c* - handles the servicing of MSMM, and NSM modules - NSM in turn services MOST NetServices, so the handling of

© 2014 Microchip Technology Inc.

all the MOST Tasks are done within this module.

- Power Management Task - *task_pmgt.c* - handles the servicing of the power management tasks including polling the MPM85000, servicing the ECLSM, TEA module, and the state machines of the power management module
- Debug Print Task - *task_print.c* - handles the background printing of the debug output if enabled

Each task is responsible for initializing its own module. This is done when called the first time from main loop. Each task has an event mechanism so the tasks can pass events to each other, the events can be used similar to semaphores of an RTOS. The tasks check for and handle these events each time they are called. For example, on the `AudioAmp` application, if the UI task detects SWA or SWB are pressed, it sets an event for the MOST Task which handles these events by changing the Volume.

Most of the functionality of these tasks has already been covered including the normal GPIO tasks of an embedded application, adding and servicing the MOST system modules (NetServices and MSMM), and the optional background debug printing. These modules described so far form the core of a MOST network application. What is needed now is the top level application logic that defines the look and feel of the overall application. For the *MOST ToGo* Basic application, that comes down to three modules:

- Power Management Module - detects why the device woke up, and whether the network should be started, and handles requests to shut-down the network. Also handles the ECL line interface with wake pulse detection and System Test support. Master devices also deal with failed startup attempts and ring breaks.
- HMI Manager - deals with receiving input from the user, deciding on the actions to take, and what should be shown on the display.
- Audio Management Module - logic between the user interface and the ConnectionMaster - determines what sources should be active and what sinks should be connected to the various sources.

3.7.1 Power Management Module

The *MOST ToGo* applications support complete MOST network power management as specified in the MOST Specification 3.0 [2], including full ECL wakeup and System Test support. The module is made up of 3 main sub modules, the power management logic module itself (*pmgt.c*), the ECL Software Module or ECLSM, which itself is divided into 5 source files, each one starting with *eclsm_*, and the System Test module, test execution application (*tea.c*). The files are in the *MOST ToGo* project virtual folder *SourceFiles/src/common/pmgt/* . As mentioned before, a future version of MSMM will incorporate the ECLSM and TEA functionality and will simplify the implementation of the power management module. The Power Management Task itself only deals with detecting the various events and periodically calling the `pmgt_service()` function which drives all of the state machines associated with the power management module.

Power management is implemented by specifying one node as the Power Master and all other nodes as Power Slaves. The Power Master node is responsible for starting and shutting down the network and initiating diagnostics (using ECL) if network lock problems occur. Power Slave nodes respond to network events and participate in wakeup and System Test sequence diagnostics.

© 2014 Microchip Technology Inc.

The PMGT module is implemented as a state machine and is executed on a regular basis in `task_pmgt`. Each state of the state machine accepts multiple events, which when received will trigger an action function that does some work, then returns whereupon the state machine may transition to another state. All of the states, events, action functions and next states are in the PMGT state machine table in the PMGT module. The PMGT state machine for a power master device has five states while power slave devices have four:

TABLE 3-1: FIVE STATES OF THE PMGT STATE MACHINE

State	Master	Slave	Description
PMGT_STATE_INIT	Y	Y	Initial state. Takes actions based on the wake reason and moves to one of the other states. This is a one-time only state.
PMGT_STATE_WAITING_NET_ON	Y	Y	The module is waiting for NetServices to indicate that the network has entered the NET_ON state.
PMGT_STATE_NET_ON	Y	Y	Normal state of operation while network is on.
PMGT_STATE_PENDING_RETRIES	Y	N	The Power Master is waiting to start network startup retries.
PMGT_STATE_POWER_DOWN	y	y	The system is in the process of powering down.

3.7.1.1 POWER MANAGEMENT STATES

A brief description of the different power management states and some of the events expected are given below.

3.7.1.1.1 PMGT_STATE_INIT

This is the initial wakeup or reset state. The events expected are the reason for the wakeup or reset, These can come from the MPM85000 (WAKE/SLEEP, ECL, STP), or from the EHC (reset, watchdog). Note that WAKE/SLEEP is the switch which goes to `ON_SW`. If the event coming in is a qualified wakeup event (WAKE/SLEEP event and ECL are the main ones), then the action function will trigger the ECL module to send out wakeup pulses on the ECL line and eventually call the MSMM Power Master API (`msm_PM_NetworkStartUp()`) to startup the network. It will then transition to the PGMT_STATE_WAITING_NET_ON_STATE. At the beginning of PMGT_STATE_INIT, the module starts its power down timer $t_{PwrSwitchOffDelay}$. If no events come in, or only events that are not considered valid wake events (STP or Switch To Power is not considered a valid event), then when the timer expires, the module will go back to the *sleep power state*.

3.7.1.1.2 PMGT_STATE_WAITING_NET_ON

In this state, the main events to wait for are NET_ON, meaning the network startup was successful and all nodes locked to the network signal, or NET_OFF meaning the startup failed. If NET_ON is received, all nodes will go to state PMGT_STATE_NET_ON. If the network failed to start, the slaves will stay in PMGT_STATE_WAITING_NET_ON, but the master device will transition to PMGT_STATE_PENDING_RETRIES.

On the master device, the action function that transitions to PMGT_STATE_NET_ON will notify the MSMM PowerMaster that NET_ON state has been reached. MSMM will in turn trigger the Network Master module to begin the system scan to check all the devices in the network and build the Central Registry.

3.7.1.1.3 PMGT_STATE_NET_ON

This is the state that all devices should eventually get to, and where they will stay during normal operation. When transitioning to this state the devices will stop their $t_{PwrSwitchOffDelay}$ shut down timers, so all devices will stay powered up while in NET_ON. The normal events that are handled here are NET_OFF, and a power down request event. When the master receives a power down event, typically the user pressing the WAKE/SLEEP switch again, then it will trigger the MSMM PowerMaster (`msm_PM_NetworkShutDown()`) to begin a shutdown sequence, then transition to PMGT_STATE_POWER_DOWN.

A NET_OFF event while in this state means that there was an unexpected break in the ring, All devices will start their $t_{PwrSwitchOffDelay}$ timers. Slave devices will transition back to PMGT_STATE_WAITING_NET_ON, while the master device will trigger network startup retries and transition to PMGT_STATE_PENDING_RETRIES.

Prior to the shutdown timer expiring, the master device can optionally trigger an ECL System Test (Stable Lock Test) which can help pinpoint the location of the break in the ring. When the system test is complete, the devices will power down.

3.7.1.1.4 PMGT_STATE_PENDING_RETRIES

Only the master device has this state, and it is reached when either a normal network startup failed, or a NET_OFF event was received while in PMGT_STATE_NET_ON. The master's PMGT module will direct the ECL module and MSMM PowerMaster to restart the network with ECL pulses and sending out network activity. When devices detect ECL or network activity, they will reset and restart their $t_{PwrSwitchOffDelay}$ shutdown timers so the devices will stay awake as long as the master device is trying to start the network. If the ring does lock and the network starts up, all devices will transition back to PMGT_STATE_NET_ON.

If the ring cannot be locked after the prescribed number of retry attempts, the PMGT module should check if a wakeup event is still valid (e.g. EngineRunning). If the wakeup event is still valid, then the PMGT module should keep triggering wakeup sequences. If there is no valid wakeup event (and the *MOST ToGo* system has no persistent wakeup events like EngineRunning), then the PMGT module will trigger an ECL System Test and perform the Stable Lock Test. The Test Execution Application (TEA module) will run the test and show the results in the LCD display. When the System Test is done, the power down timers are started, and the devices will all revert back to the *sleep power state*. It is up to the OEM whether or not to trigger this System Test at the end of failed startup sequences.

3.7.1.1.5 PMGT_STATE_POWER_DOWN

This state is reached in a master device when a power down request event is received while the master is in PMGT_STATE_NET_ON. On the *MOST ToGo* system, this event is caused by the user pressing the WAKE/SLEEP switch. The master's PMGT module will call the MSMM PowerMaster module (via `msm_PM_NetworkShutdown()`)to start the shutdown sequence and then transition to this state.

Slaves will transition to this state from PMGT_STATE_NET_ON when they receive the *Netblock.Shutdown.Query* message from the PowerMaster. If the slave has no reason to stay powered up, it should save its settings and transition to this state waiting for NET_OFF.

© 2014 Microchip Technology Inc.

While in this state, all devices are essentially waiting for NET_OFF. As part of the shutdown sequence (when no devices respond to the *Netblock.Shutdown.Query* message), the PowerMaster will turn off the network signal, and all devices will detect the NET_OFF event, and will start their power down timers $t_{PwrSwitchOffDelay}$. When the timers expire all devices will shut down, reverting back to the *sleep power state*.

3.7.2 ECL Software Module (ECLSM)

The *MOST ToGo* PMGT module includes the ECLSM, although as mentioned previously this will soon become part of MSMM. The ECLSM module handles the following actions to satisfy the requirements of the MOST Electrical Control Line Specification [11]:

- Detect and qualify wakeup sequences
- Detect and qualify System Test Sequences
- Extract parameters from System Test Sequences
- Generate wakeup sequences (Master and Slaves that support slave wakeup)
- Generate system test sequences (Master only)
- Report the results of System Tests

The ECLSM runs continuously to monitor the ECL line and reports wakeup or test sequences to the PMGT module, where they are turned into events to the PMGT state machine. These events can also be passed on to other application modules if needed. The PMGT module also calls the ECLSM to initiate wakeup and test sequences under various conditions. Test sequences are handled automatically by the ECLSM and the TEA module handles execution of each particular test required by the ECLSM.

The Power Master node may be a wakeup/test initiator or participant. Power Slave nodes are participants only. An external test tool connected to the ECL may initiate wakeup and system test sequences, in which case the Power Master and Slaves are all participants.

Each node has its own ECL System Test Node Class and reports its test results in the appropriate time slot of the System Test result phase. The Power Master node uses Node Class 0, the Auxin node uses Node Class 1, and the Amp node uses Node Class 2. The rotary switch S1 is used to set the Node Class uniquely for each board.

3.7.3 Test Execution Application (TEA Module)

Once the ECLSM detects that some device is signaling a System Test on the ECL interface and has extracted the parameters to determine which test to perform, it will call the TEA module to actually execute the test. The *MOST ToGo* applications support the following tests which are defined in the MOST Electrical Control Line Specification [11].

- Ring Break Diagnostics (RBD)
- Device Alive Test
- MOST Coding Errors Test
- Stable Lock Test

Once the indicated test is performed by the TEA module, it will report the result back the ECLSM which will report the result in its proper time slot.

© 2014 Microchip Technology Inc.

3.7.4 HMI Module

As mentioned in section Section 2.2.7 “Human Machine Interface (FBlock HMI)”, the user interface provides the look and feel of the whole application, and is the domain of the OEM or selected Tier 1. Therefore not much detail will be provided on the implementation of the *MOST ToGo* user interface, as it is unlikely to be used in a production environment. The focus will be on how user actions trigger the application to carry out the requested action.

The user interface for the basic *MOST ToGo* system is made up of the LCD display, the RightTouch buttons, FBlock `HMI`, the HMI_Manager, and the OSD (On Screen Display) module. The OSD module is a general purpose menu system for text displays like terminals, and the LCD display of the *MOST ToGo* system. Think of the OSD as a state machine, and the tables in the HMI_Manager as the state tables. The OSD module itself is generic in that it can support any number of lines of display, and any number of key inputs. For this system it is setup for the 4 line display and 7 input keys. The OSD system supports the navigation and drawing of the menus and can activate functions when certain keys are pressed at the terminal point of a menu. The menus, key definitions, and callback functions are defined in the HMI_Manager module.

The menu system can be thought of as a hierarchical file structure with some menu items leading to new menus or back to previous menus (branches), while others are terminal items (leafs). The menu navigation is handled completely in HMI_Manager and the OSD module. When the current menu line is a ‘leaf’ and either the Left, Right, or Select buttons are pressed (context sensitive), then a callback function is fired.

The best way to see how this all works is to go step by step through an example. Consider that the menu system is displaying a menu that shows the current Volume. The user moves to the Volume line and presses either the Left or Right arrow button. The action should be to decrease or increase the volume.

- User presses Right Arrow button
- Press is detected by `task_ui` which is polling the Interrupt line connected to the RIghtTouch device.
- `drv_touch_event_handler()` is called by `task_ui` to handle the interrupt signal
- `drv_touch_event_handler()` reads the RightTouch device and determines which button was pressed
- `drv_touch_event_handler()` fires a button event callback with the button number as a parameter
- That callback is `button_event_handler()` in `task_ui` which builds a MOST network message to FBlock `HMI`.
- `HMI.ButtonStatus.Set(button)`
- FBlock `HMI` receives message and calls `hmim_process_key(button)` which is in HMI_Manager
- `hmim_process_key()` simply calls `ServiceMenu()` in the OSD module
- The `ServiceMenu()` function will look in the current menu tables and see that on a Right Key press it should activate the function `cb_on_change_sink_volume()` which is in HMI_manager.
- The `cb_on_change_sink_volume()` function will build a MOST net-

© 2014 Microchip Technology Inc.

work message to FBlock AudioAmp, getting the InstID from a table created by the AV_Manager from information supplied by the Connection Master.

- Amp.InstID(sink_under_control).Volume.OP_INC

NetServices will add the physical node address to the message based on FBlock.InstID and send the message out.

This is the end of the sequence as far as the HMI_Manager is concerned. The display is not updated until a reply is received from the AudioAmp that the volume was changed. That event gets handled by FBlock AudioAmp Shadow which links to the AV_Manager which will be seen in the next section.

3.7.5 Audio/Video Management Module

The AV_Manager module (*avm.c*) has the logic that is between user interface commands, the Connection Master, and the FBlocks on the network. The module in the *MOST ToGo* system is technically only an audio manager since there is no video in the basic system. But, the control principles are the same, and the system will be expanded in future application examples to include video, and the AV_Manager module will be expanded as well. This module should have the logic that decides what should be connected based on user input, and then it will send messages or call functions to create the connections.

Besides this control logic, the AV_Manager has a second major function in that it also serves as the shadow for FBlocks AuxIn and AudioAmp. The sections below describe the control aspect and the shadow aspect of the AV_Manager.

3.7.5.1 AV_MANAGER AS CONTROLLER

In the Volume example in the previous section, the HMI_Manager built a network message to send directly to the required FBlock. If the command had been to connect a source to a sink, then the HMI_Manager will call the AV_Manager (avm_connect()) to do the work. The HMI_Manager will pass to the AV_Manager which source and sink should be connected.

The AV_Manager will look up the source and sink to determine their FBlock and InstID's which are needed by FBlock ConnectionMaster. The AV_Manager has the option to either send a message to FBlock ConnectionMaster, or since it is on this same device with the AV_Manager, it can build a direct function call to the Connection Master (msm_CM_BuildConnection()). In the *MOST ToGo* implementation, it builds a direct function call, but there is a build option to send a message instead. If the Connection Master had been located on a different device across the network, then the AV_Manager would have had to build a message to send to FBlock ConnectionMaster.

Once activated, whether by direct function call or by receiving a network message, the Connection Master will use its connection tables to send the messages to the required devices, commanding the source to *Allocate* a MOST network channel and stream its data, then commanding the sink to *Connect* to that network channel after it has been established.

In the basic *MOST ToGo* system there are only a few possible simple connections, so not much logic is needed. Request from the HMI Module are simply passed to the AV_Manager where they are carried out. In a more typical system, there may be priorities associated with connections. For example, if a request is received to listen to FM radio, but there is currently a connection to the phone, then the phone should have higher priority and the radio request is

© 2014 Microchip Technology Inc.

either ignored or perhaps queued and acted on once the phone is disconnected. That type of logic would be in the AV_Manager.

Also with more sophisticated devices, the suitability of connections must be considered. The AV_Manager should make sure the requested connections are valid, for example if it somehow gets a request to connect a video source to an amplifier, or to connect an encrypted 5.1 audio stream to a stereo amplifier that does not support DTCP decryption. All of the needed information to determine the validity of connections should be in the shadow structure maintained by the AV_Manager. In these examples the data types of the streams must match before a connection is allowed.

3.7.5.2 AV_MANAGER AS SHADOW

As mentioned above, the AV_Manager is also the shadow for FBlocks `AuxIn` and `AudioAmp`. The AV_Manager subscribes to all notification that FBlocks `AuxIn` and `AudioAmp` support, and must be able to handle the resulting messages. A look at the command interpreter tables for FBlock `AuxIn` Shadow or FBlock `AudioAmp` Shadow will show that all of the status replies map to functions in the AV_Manager module. See Example 3-19 OpType Table for the Volume Property of FBlock AudioAmp Shadow

EXAMPLE 3-19: OpType TABLE FOR THE VOLUME PROPERTY OF FBlock `AudioAmp` SHADOW

```
//-------------------------------------------------------------------------------------------------------//
//  Table of available Operations of     FBlock:   Amplifier                                             //
//                                       Function: Volume                                                //
//-------------------------------------------------------------------------------------------------------//
//  OP_TYPE           |  Flags |     Ptr for write access  |   Ptr for read access    | Length Check  //
//-------------------------------------------------------------------------------------------------------//
                                                                                                        //
const struct Op_L_Type Op_Amp_Shdw_Volume[] =                                                           //
  {                                                                                                     //
    {OP_STATUS,        0x50,    NS_F_V avm_shadow_amp_volume_status,    0,            LC_NO   },    //
    {OP_TERMINATION,   0,       0,                                     0,            LC_NO         //
};
//-------------------------------------------------------------------------------------------------------//
```

Any message from FBlock Amp with an OpType of OP_STATUS will get mapped to a function in the AV_Manager. The AV_Manager maintains a data structure where all the data from the shadow functions are saved. For the `avm_shadow_amp_volume_status()` function from the command interpreter table above, the main work of the shadow function is done in the following 2 lines of code (Example 3-20 Reacting to Property Changes):

EXAMPLE 3-20: REACTING TO PROPERTY CHANGES

```
        avm.source_sink_list[index].data[AVM_AMP_VOLUME] = Rx_Ptr->Data[0];
        hmim_on_volume_change(Rx_Ptr->Data[0], Rx_Ptr);
```

The first line saves the Volume value sent in the status message in the data structure. The second line calls the HMI Module with the new value to possibly

© 2014 Microchip Technology Inc.

update the display. Prior to saving the data, a function is called to convert the FBlock and InstID of the message into the `index` value. The `index` points into the data structure, and each FBlock and InstID of the discovered sinks and sources in the system will have an entry. Since the AVModule subscribes to notification of all sinks and sources in the system, the table will get fully populated with the current values of all properties being monitored. In other words, the AV_Manager's data structure (`source_sink_list`) will end up containing a *shadow* of all the properties of all the sinks and sources.

When a new property comes in to the shadow, the AV_Manager may also call the HMI Module with the new value as seen above. This keeps the display in sync with the current values of all the properties. The HMI Manager will in turn update its internal display menus, and if the property is currently being displayed, then the value will update in the display.

© 2014 Microchip Technology Inc.

Appendix A. Glossary and General Terms

TABLE A-1: GLOSSARY

Term	Acronym	Definition
Active Power State		ECU state in which the device is connected to a continuous battery supply and fully active. EHC and INIC are powered. This state does not automatically indicate that the MOST network is operational. The opposite of *active power state* is *sleep power state*.
Application Programming Interface	API	Specifies how some software components should interact with each other.
Audio Amplifier	AMP	Device that increases the amplitude of a signal.
Battery Continuous Power	BatConP	Power management conceptual power supply net which is always connected directly to the battery. This net can have different names based on where the signal is viewed. BatConP is $V_{BATTERY}$ at the battery terminals, and V_{BAT_ECU} at an ECU power connector; the difference being the voltage drop that occurs in the wiring between the two points.
Battery Switched Power	BatSwP	Power management conceptual power supply net which is controlled by the power-master ECU and provides power to all other network ECUs. In this scenario, the power-master ECU must be on BatConP.
Bypass Mode		INIC state where the incoming network data is bypassed directly to the TX output of INIC. While in bypass mode, INIC is invisible to the network and cannot transmit any data. The electrical MOST50 INIC does not support bypass mode. The opposite of *bypass* mode is a *visible node*.
Central Registry		Contains a lookup table for cross-referencing logical and functional addresses. Implemented in the network master.
Clamp Status		Indication of the ignition key position (off, accessory, on).
Coaxial Physical Layer	cPhy	Coax network cable used in MOST150 networks.
Coaxial to Electrical Converter	CEC	Devices designed for MOST150 networks using a coax PHY layer to receive coax signals and convert them to electrical signals for INIC to use. Includes circuitry for sleep mode support.
Coaxial Transceiver	CTR	Standard MOST network connector unit for a MOST150 cPHY network connection consisting of a CEC and ECC.
Connection Manager		Manages streaming connections.
ConnectionMaster		FBlock, which is an interface to the connection manager.
Controller Area Network	CAN	A vehicle bus standard designed to allow microcontrollers and devices to communicate with each other within a vehicle without a host computer.
Decentral registry		Contains a lookup table for determining available function blocks and cross-referencing logical and functional addresses.
Device		Physical unit, which can be connected to the MOST network via a MOST Network Interface Controller.

© 2014 Microchip Technology Inc.

TABLE A-1: GLOSSARY (CONTINUED)

Term	Acronym	Definition
Diagnostics Over Internet Protocol	DoIP	ISO 13400-2:2012 specifies the requirements for diagnostic communication between external test equipment and vehicle electronic components using Internet Protocol (IP) as well as the transmission control protocol (TCP) and user datagram protocol (UDP). This includes the definition of vehicle gateway requirements (e.g. for integration into an existing computer network) and test equipment requirements (e.g. to detect and establish communication with a vehicle).
Electrical to Coaxial Converter	ECC	Devices designed for MOST150 networks using a coax PHY layer to convert INIC electrical signals into coax signals for transmission on the MOST network.
Electrical Control Line	ECL	Wire-OR'd unshielded cable attached to every ECU. Defined by the MOST Cooperation in the Electrical Control Line Specification] and used for wakeup as well as diagnostic communications.
ECL System Test Start Pulse	TS	
ECL Wakeup Pulse	WI	
Electronic Control Unit	ECU	Also known as a "device" in the MOST Specification 3.0 [2]. An entire box or unit consisting of an INIC, EHC, power management circuitry and application circuitry.
Electrical Optical Converter	EOC	Produces a digital signal in the electrical domain from an incoming optical signal.
Electrical Physical Network	ePhy	Unshielded twisted pair network cable used in MOST50 networks. ePhy network standards can be found in the MOST Electrical Physical Layer Specification.
Electrical Wakeup	EWU	
EnhancedTestability	ET	
External Host Controller	EHC	Microcontroller or microprocessor that manages the ECU and defines what applications exist.
Function		Part of an FBlock through which it communicates with the external world.
Function Block	FBlock	Logical group of functions (commands) that are related. For example, an FBlock Tuner contains all functions associated with the radio tuner hardware. All FBlocks are required to support some common functions for plug-and-play operation. In addition, all ECUs are required to support some standard FBlocks such as NetBlock (also used for plug-and-play operation/system enumeration). Some network system services also reside in FBlocks, such as the FBlock `NetworkMaster`. See the Device Model Section 2.1.2 in the MOST Specification 3.0 [2].
Fiber Optic Receiver	FOR	Also known in the MOST Specification 3.0 [2] as optical electrical converter, OEC. Devices designed for MOST networks using an optical PHY layer to receive optical signals and convert them to electrical signals for INIC to use. Includes circuitry for sleep mode support.
Fiber Optic Transceiver	FOT	Standard MOST network connector unit for an oPHY MOST network consisting of an FOR (OEC) and FOX (EOC).

TABLE A-1: GLOSSARY (CONTINUED)

Term	Acronym	Definition
Fiber Optic Transmitter	FOX	Also known in the MOST specification as electrical optical converter, EOC. Devices designed for MOST networks using an optical PHY layer to convert INIC electrical signals to optical signals.
Firmware	FM	The control program for a device, consists of the combination of persistent memory and program code and the data stored in it.
Ground	GND	Refer to the reference point in an electrical circuit from which voltages are measured, a common return path for electric current, or a direct physical connection to the Earth.
Head Unit	HU	Typically a component of the infotainment network which provides a unified hardware interface for the various components of an electronic media system.
INIC Control Message	ICM	
Intelligent Network Interface Controller	INIC	Manages all time critical low-level network functions to off load the EHC and provide a more stable network. Protects the network from errant EHC code. INICs have been developed for all MOST system speed grades and provide simple migration paths between grades.
International Organization for Standardization	ISO	An international standard-setting body composed of representatives from various national standards organizations.
Local Interconnect Network	LIN	A low-cost low-speed automotive network managed by the LIN Consortium (www.lin-subbus.org). LIN transceivers can be used to implement the MOST Electrical Control Line (ECL).
Local Wakeup Event		Event on an ECU that is unique to the ECU. This event must be qualified (cannot be a glitch or power supply change) before the ECU can generate a network wakeup event, waking the rest of the network ECUs. Examples of local wakeup events include getting clamp status (key position) from a separate network (such as CAN), an ON switch being pressed (which must be qualified first - debounced), a wireless receiver receiving a call (which must be qualified - verify that the rest of the network is needed based on the call received).
Media Local Bus	MediaLB® or MLB	Board-level high-speed bus that connects INICs to EHCs and other peripherals that can carry all MOST network data types.
MOST® Control Message	MCM	
MOST® Data Packet	MDP	
MOST® Ethernet Packet	MEP	
MOST® High Protocol	MHP	A connection-oriented protocol using some of the mechanisms of the TCP protocol. Located between Network Services and the function blocks of a MOST device.
Media Oriented Systems Transport	MOST®	High-speed networks, designed for automotive use, that efficiently carry streaming data (audio/video), network control data and packet data. MOST150 also carries multiple types of isochronous data as well as Ethernet packet data. The MOST network standard is managed by the MOST Cooperation (www.mostcooperation.com). MOST25 - is a 25 Mbit/s automotive bus MOST50 - is a 50 Mbit/s automotive bus MOST150 - is a 150 Mbit/s automotive bus that supports extra data types

© 2014 Microchip Technology Inc.

TABLE A-1: GLOSSARY (CONTINUED)

Term	Acronym	Definition
Maximum Position Register	MPR	Provides the Network Master with information about the number of nodes in the system that it must query.
MOST® System Management Module	MSMM	Handles system-related tasks “above” the MOST NetServices API and can be considered as “middleware” of a MOST System.
NetInit State		The NetInit state corresponds to the “NetInterface Init Operation” state.
NetInterface		State of the EHC communication with respect to the rest of the network. Defined in the NetInterface Section 3.1.2.2 of the MOST Specification 3.0 [2].
NetOff State		The NetOff state corresponds to the “NetInterface Off Operation” state.
NetOn State		The NetOn state corresponds to the “NetInterface Normal Operation” state.
Network Interface Controller	NIC	Network Interface Controller. Integrated circuits that implement the original MOST25 speed grade, and require all network startup intelligence to be managed by the EHC.
Network Change Event	NCE	
Network Master	NWM	ECU containing the FBlock `NetworkMaster` software. Controls the system state and administrates the central registry of all FBlocks on the network and their current state. Defined in the Network Master Section 3.1.3.3 of the MOST Specification 3.0 [2].
Network Wakeup Event		Event intended to wake (from *sleep power state*) all network ECUs. Network wakeup events include network activity, ECL assertion or a power supply STP pulse. Opposite of a local wakeup event.
Node Class		Occupies a result slot in the ECL test result sequence. Every device is associated with exactly one node class. The more node classes are defined, the longer the ECL test will last. Within one system, a node class must not be assigned to more than one device. The node class may be identified by the system integrator.
Original Equipment Manufacturer	OEM	A company who manufactures products or components that are purchased by another company and retailed under that purchasing company's brand name.
Optical Electrical Converter	OEC	Also know as fiber optic receiver, FOR. Devices designed for MOST networks using an optical PHY layer to receive optical signals and convert them to electrical signals for INIC to use. Includes circuitry for sleep mode support.
Optical Physical Network	oPhy	Network connections that use FOTs and plastic optic fiber for ECU-to-ECU connections.
Polymer Optical Fiber	POF	A 1 mm diameter plastic fiber used in MOST25 and MOST150 optical physical networks.
Power Master		Logical software block which manages power up and power down of the network. Traditionally resides on the ECU which contains the timing master. Only one power master can exist in the MOST network. All other ECUs are designated as power slaves.

© 2014 Microchip Technology Inc.

TABLE A-1: GLOSSARY (CONTINUED)

Term	Acronym	Definition
Ring Break Diagnosis	RBD	Diagnosis mode built into INIC chips to help determine where a break exists in the MOST network. Defined in the MOST Specification 3.0 [2] for MOST50 and MOST150.
Rear-Seat Entertainment System	RSE	Infotainment systems within a vehicle that reside behind the front seat cabin.
Sleep Power State		ECU state in which the device is connected to a continuous battery supply, but is drawing minimal current (I_{STBY}). Most of the circuitry in the ECU is powered off; only circuitry needed to wake up is powered. Maximum *sleep power state* current is generally specified by the system integrator or OEM. The opposite of *sleep power state* is *active power state*.
Software	SW	Any set of machine-readable instructions that directs a computer's processor to perform specific operations.
Switched-Application Signal	SA	Optional digital signal from the EHC that controls power to the applications switched power block of circuitry. This signal allows the EHC to power down the application circuitry while still keeping the network alive.
Switch-To-Power	STP	Method to start Ring Break Diagnosis by removing battery power for greater than 2 s. This method is deprecated due to the complexity of synchronizing all ECUs regarding the power event.
System Integrator		Entity responsible for coordination of all devices in the network from all suppliers.This entity could be the OEM, or an ECU supplier designated by the OEM.
System Scan		Process of collecting information from the network slaves, performed by the network master.
System State		Two System States are possible: OK and NotOK. In OK, the system is in normal operation mode. In NotOK, the system is being initialized or updated.
Timing Master	TM	INIC that is configured as the master clock for the network. The timing master INIC generates the system clock and framing signals for the network. All other network ECUs are designated as timing slaves, which synchronize to the incoming network clock.
User Interface	UI	The space where interaction between humans and machines occurs.
Visible Node		Normal INIC state where INIC is visible to all other network ECUs, has a node position, and can communicate with all other ECUs. The opposite of a visible node is bypass mode.

© 2014 Microchip Technology Inc.

NOTES:

Worldwide Sales and Service

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://www.microchip.com/support
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Austin, TX
Tel: 512-257-3370

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Cleveland
Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Novi, MI
Tel: 248-848-4000

Houston, TX
Tel: 281-894-5983

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

New York, NY
Tel: 631-435-6000

San Jose, CA
Tel: 408-735-9110

Canada - Toronto
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2943-5100
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8569-7000
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Chongqing
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

China - Hangzhou
Tel: 86-571-8792-8115
Fax: 86-571-8792-8116

China - Hong Kong SAR
Tel: 852-2943-5100
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8864-2200
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-3019-1500

Japan - Osaka
Tel: 81-6-6152-7160
Fax: 81-6-6152-9310

Japan - Tokyo
Tel: 81-3-6880- 3770
Fax: 81-3-6880-3771

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-5778-366
Fax: 886-3-5770-955

Taiwan - Kaohsiung
Tel: 886-7-213-7830

Taiwan - Taipei
Tel: 886-2-2508-8600
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Dusseldorf
Tel: 49-2129-3766400

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Germany - Pforzheim
Tel: 49-7231-424750

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Italy - Venice
Tel: 39-049-7625286

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Poland - Warsaw
Tel: 48-22-3325737

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

Sweden - Stockholm
Tel: 46-8-5090-4654

UK - Wokingham
Tel: 44-118-921-5800
Fax: 44-118-921-5820

03/25/14

© 2014 Microchip Technology Inc.